jamk.fi

Automaattisten ylläpitotestien toteutus

Veetu Salminen
Santeri Moilanen

Opinnäytetyö
Toukokuu 2018
Liiketalouden ala
Tradenomi (AMK), tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijät	Julkaisun laji	Päivämäärä
Salminen, Veetu Moilanen, Santeri	Opinnäytetyö, AMK	Toukokuu 2018
	Sivumäärä 62	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Automaattisten ylläpitotestien toteutus

Tutkinto-ohjelma
Tietojenkäsittelyn tutkinto-ohjelma

Työn ohjaaja
Niko Kiviaho

Toimeksiantaja
Solteq

Tiivistelmä

Testaus on merkittävässä roolissa ohjelmistokehityksessä. Hyvin toteutettu testaus parantaa ohjelmiston laatua ja auttaa ohjelmistovirheiden havaitsemisessa. Testausta voidaan tehdä eri ohjelmistokehityksen vaiheissa. Se on tärkeää myös ohjelmiston ylläpitovaiheessa.

Tutkimuksen tavoitteena oli selvittää, miten verkkokauppaan olemassa olevat ylläpitotestit saadaan toimimaan jälleen. Lisäksi tavoitteena oli tehdä testeistä mahdollisimman laadukkaat ja asettaa ne suoriutumaan automaattisesti. Tutkimus toteutettiin kehittämistutkimuksena, sillä siinä pyrittiin kehittämään olemassa olevia ylläpitotestejä. Testausvälineinä käytettiin avainsanapohjaista Robot Frameworkia sekä testien automatisoinnissa Jenkins-ohjelmistoa.

Opinnäytetyön tuotoksena valmistuivat Robot Frameworkilla toteutetut ylläpitotestit, jotka on automatisoitu Jenkinsin avulla. Ylläpitotesteistä tehtiin mahdollisimman helposti ylläpidettävät ja suorituskykyiset. Testit asetettiin suoriutumaan automaattisesti joka yö tuotantoon viennin jälkeen. Lisäksi testitapauksista tehtiin dokumentti, jossa on kuvaukset testitapauksista. Toteutetut testit ovat hyödyttäneet verkkokaupan kehitystiimiä ohjelmistovirheiden löytämisessä sekä säästäneet aikaa manuaaliselta testaustyöltä.

Työn tuloksena saatiin tehtyä yritykselle tietopaketti siitä, miten Robot Frameworkilla toteutetaan ja ylläpidetään testejä. Lisäksi työstä käy ilmi, miten testit saadaan asetettua osaksi Jenkinsiä. Testit toteutettiin PO-käytänteiden mukaisesti, mikä poikkeaa Robot Frameworkin avainsanapohjaisesta tyylistä. Tämä toteutustapa todettiin toimivaksi, etenkin sellaisessa tilanteessa, jossa testitapauksia on useita.

Avainsanat (asiasanat)

ohjelmistotestaus, ylläpitotestaus, Robot Framework, Jenkins, jatkuva integraatio

Muut tiedot

jamk.fi

Description

Authors	Type of publication	Date
Salminen, Veetu Moilanen, Santeri	Bachelor's thesis	May 2018
		Language of publication: Finnish
	Number of pages 62	Permission for web publication: x

Title
Implementation of automatic maintenance tests

Degree programme
Business Information Technology

Supervisor
Kiviaho Niko

Assigned by
Solteq

Testing has a notable role in software development. Well implemented testing improves the quality of the software and helps to detect errors in the software. Testing can and should be done in many different stages of software development. It also plays a prominent role in the maintenance stage of the software.

The main objective of this thesis was to investigate, how the current maintenance tests can be made to function again. Additionally, these tests needed to be more comprehensive and run automatically. This thesis is a design-based research, since the main objective was to improve the functionality of the current maintenance tests. Testing tools used for this were keyword-based Robot Framework and Jenkins software was used for the automation process.

The output from this thesis were the maintenance tests made with Robot Framework, which were automated using Jenkins software. The maintenance tests were improved on to make them more maintainable and efficient. The tests were set up to be run every night after product deployment was done. In addition, a test documentation was made with descriptions of every test case. The tests currently in production have helped the development team's ability to find defects in software and save time from doing manual testing work.

As a result of the project, the enterprise was provided with an information package on how to implement and maintain tests with Robot Framework. In addition, the work shows how tests can be automated using Jenkins. The tests were carried out in accordance with PO practices, which differ from Robot Framework's keyword-based style. This implementation style was found to be practical; especially in situations where there are multiple test cases.

Keywords/tags (subjects)
software testing, maintenance testing, Robot Framework, Jenkins, continuous integration

Miscellaneous

Sisältö

Kuviot

Käsitteet ja termit

Apache 2.0 lisenssi	Vapaan ohjelmiston lisenssi, joka sallii lähdekoodin vapaan ja avoimen käytön.
Avoin lähdekoodi	(engl. Open Source) Lisensointi, jossa ohjelmakoodi annetaan tietyin rajoituksin levitettäväksi ja muokattavaksi.
Deploy	Joukko toimenpiteitä, joiden avulla sovellus tuodaan käyttäjien saataville.
Debuggeri	(engl. debugger) Ohjelma, jonka avulla jäljitetään virheitä ohjelmakoodissa.
Git	Versionhallintajärjestelmä
Hipchat	Ryhmäkeskustelukanava työkäyttöön
HTML	(engl. Hypertext Markup Language) Avoimen standardin merkintäkieli, jota käytetään pääasiassa web sivuilla.
IDE	Integrated development environment. Suomeksi integroitu kehittämisympäristö. Se sisältää tyypillisesti tekstieditorin, kääntäjän ja debuggerin.
Java	Laajassa käytössä oleva oliopohjainen ohjelmointikieli.
Jenkins	Web-pohjainen jatkuvan integraation sovellus, jolla voidaan tarkkailla ja konfiguroida toistuvasti suoritettavia tehtäviä
Kommitoida	Muutosten päivitys versionhallintajärjestelmään.
Käännös	Ihmiselle helpossa muodossa olevan ohjelmointikielen muunnos tietokoneen ymmärtämään muotoon.
LTS	Long Term Support. Versio ohjelmasta, johon tulee päivityksiä harvemmin kuin ohjelman perus versioon.
Mercurial	Versionhallintajärjestelmä
Pip	Python ohjelmointikielen paketinhallintajärjestelmä

PO	Page-Object, kuvaa yhtä sivun loogista kokonaisuutta, esim. Ostoskori tai navigaatio palkki.
Python	Helppolukuinen, monipuolinen ja tulkattava ohjelmointikieli.
Responsiivisuus	Sivujen skaalautuminen eri laitepäätteillä.
Robot Framework	Yleinen testiautomaatio sovelluskehys hyväksyntätestaukseen.
Selenium	Sovelluskehys web-sovellusten testaamiseen.
Sovelluskehys	Ohjelmistotuote, joka muodostaa pohjan sen päälle rakennettavalle ohjelmistolle.
TCP/IP	(engl. Transmission Control Protocol / Internet Protocol) Kahden protokollan yhdistelmä, jolla kaksi päätelaitetta voi lähettää tietoja toisilleen.
Trigger	Suomeksi laukaisin. Tekee, jonkun toimenpiteen tiettyjen ehtojen täytyttyä. Esimerkiksi näytä teksti, kun käyttäjä painaa näppäintä.
Versionhallinta-järjestelmä	(engl. version control system) Järjestelmä, jonka avulla pidetään kirjaa tiedostoissa tapahtuvista muutoksista. Sen avulla voidaan palata tiettyyn versioon myöhemmin.
War-tiedosto	(engl. Web Application Archive) Tiedostomuoto Java-pohjaisille web sovelluksille.
XML	(engl. Extensible Markup Language) Rakenteellinen kuvauskieli, käytetään formaattina tiedonvälityksessä.
Xpath	XML elementtien valintaan käytettävä kyselykieli.

1 Johdanto

Testaaminen on merkittävä osa ohjelmistokehitystä. Sillä on suora vaikutus ohjelmiston laatuun. Mitä vähemmän ohjelmistosta löytyy virheitä sen tyytyväisempiä asiakkaat ja itse käyttäjät ovat. Virheiden korjaaminen jälkikäteen on yleensä kallista ja se johtaa helposti aikataulujen venymiseen. Testien automatisoinnilla pyritään lisäämään testauksen kattavuutta ja vapauttamaan kehittäjiltä työaikaa itse tuotteen kehitykseen. Automatisoinnilla pyritään varmentamaan, ettei uusia ominaisuuksia lisätessä vanhat toiminnot hajoa.

Opinnäytetyössä on tavoitteena toteuttaa automaattinen ylläpitotestaus asiakkaan verkkokauppaan. Asiakkaan verkkokauppaan on olemassa vanhentuneet ylläpitotestit, mutta ne eivät ole tarpeeksi kattavat ja ne eivät toimi oikein. Tutkimuksessa pyritään tuomaan esille, kuinka testit korjataan ja miten niiden rakenne muutetaan selkeämmäksi. Lisäksi tavoitteena on tehdä testeistä nopeammat ja rakenteeltaan selkeämmät. Testit asetetaan myös suoriutumaan automaattisesti.

Tutkimusasetelmassa käydään läpi lähtötilanne, toimeksiantaja, selvitetään työn tavoitteet, rajataan aihe ja selvitetään käytettävä tutkimusmenetelmä. Lisäksi käydään läpi, mitä konkreettista hyötyä toimeksiantaja saa opinnäytetyöstä. Työn teoria osassa tutkitaan testaamista yleisesti, käydään läpi automaattisen testauksen periaatteet sekä tutustutaan työssä käytettyihin työvälineisiin. Varsinaisessa toteutus osassa korjataan vanhat testit ja parannellaan niiden rakennetta. Tämän jälkeen testejä laajennetaan ja ne automatisoidaan. Lopuksi esitetään tutkimuksen tulokset tutkimuskysymyksiin verraten ja pohditaan työhön vaikuttaneita tekijöitä.

2 Tutkimusasetelma

2.1 Tausta, tavoitteet ja rajaus

Toimeksiantajalla on tarve saada vanhat Robot Frameworkilla toteutetut ylläpitotestit toimimaan jälleen asiakkaan verkkokaupassa. Testit ovat ylläpitotestien lisäksi savutestejä eli niillä varmennetaan verkkokaupan perustoimintoja, kuten sisäänkirjau-

tumista. Lisäksi toimeksiantajalla on tarve tehdä testeistä ylläpidettävämpiä, nopeampia sekä kattavampia. Toimeksiantajan pyynnöstä asiakas pidetään työssä salassa ja tästä eteenpäin asiakkaasta käytetään termiä asiakas X.

Tavoitteena on saada korjattua vanhat ylläpitotestit sekä tarkoitus on tehdä niistä ylläpidettävämpiä, selkeämpiä ja nopeampia. Tavoitteena on myös lisätä testien kattavuutta. Hyötynä tästä seuraa kehitystiimin saama jatkuva palaute automaattisista testeistä, joka auttavat virheiden löytämisessä ja täten parantavat tehokkuutta ja verkkokaupan laatua. Hyvä laatuiset ja riittävän laajat testit vähentävät projektiin käytettyjä työtunteja ja täten lisäävät kustannustehokkuutta.

Lisäksi tavoitteena on saada asetettua testit ympäristöön, jossa niitä suoritetaan automaattisesti. Tähän tarkoitukseen toimeksiantajalla on olemassa testipalvelimia, joille on asennettu automaation toteuttava työkalu Jenkins.

Automaattisten ylläpitotestien toteutus sekä testien automatisointi rajataan siten, että ne toteutetaan samoilla testausvälineillä, mitä toimeksiantajalla on käytössä. Testaus rajoittuu ylläpitotestien korjaukseen ja parantamiseen. Testeistä ei ole tarkoitus tehdä täysin kattavia ylläpitotestejä vaan perusasioita testaavia savutestejä. Testit kuitenkin toteutetaan siten, että niiden määrää on myöhemmin helppo kasvattaa, mikäli niistä halutaan tehdä kattavampia. Testiautomaation osalta käydään läpi, miten testit asetetaan osaksi Jenkins automaatio-putkea. Tarkoitus ei ole syventyä tarkemmin siihen, miten Jenkins asennetaan ja konfiguroidaan, vaan ainoastaan asettaa testit osaksi tätä testausjärjestelmää.

2.2 Tutkimuskysymykset

Opinnäytetyö pyrkii vastaamaan seuraaviin tutkimuskysymyksiin. Ne ovat johdettu tutkimuksen tavoitteiden ja toimeksiantajan tarpeiden mukaisesti.

Tutkimuskysymykset:

- Miten automaattinen ylläpitotestaus toteutetaan asiakkaan X verkkokauppaan?
- Miten Robot Frameworkilla kirjoitetaan selkeitä, ylläpidettäviä ja nopeita testejä?
- Kuinka testit saadaan integroitua osaksi Jenkinsin automaatioputkea?

2.3 Toimeksiantaja

Solteq on keskisuuri suomalainen kaupan, logistiikan ja ohjelmistopalveluiden suunnitteluun ja valmistukseen perehtynyt yritys. Solteq on digitaalisen asiakaskohtamaisen asiantuntija ja on erikoistunut tuottamaan erilaisia kaupan alan palveluja asiakkailleen. Yhtiön asiakkaat ovat pääosin kaupanalan yhtiöitä ja Solteq työllistää tällä hetkellä noin 550 asiantuntijaa viidessä eri maassa.

2.4 Tutkimus-/kehittämismenetelmät

Yritykset pyrkivät jatkuvasti kehittämään ja parantamaan toimintaansa. Kehittämistutkimus on eräs keino kehittämistyön tulosten julkituomiseksi. (Kananen 2015, 33.) Opinnäytetyön toteutetaan kehittämistutkimuksena, sillä siinä pyritään kehittämään olemassa olevia ylläpitotestejä.

Kehittämistutkimuksessa yhdistyy nimensä mukaisesti kehittäminen ja tutkiminen syklisessä prosessissa. (mts. 33.) Kanasen (2015, 33) mukaan ”Syklinen prosessi tarkoittaa ns. kehittämissykliä, jossa kuvataan ongelma, laaditaan toimenpide-ehdotukset, toteutetaan ja katsotaan tulokset (vrt. toimintatutkimuksen sykliin).” Kehittämistyön lähtökohtana on aina tarve muutokseen, eli siinä tavoitellaan kehitystä olemassa olevaan tilaan. Jotta kehitystyö olisi kehittämistutkimusta tulee siihen sisällyttää tutkimus ja sen tulosten ja prosessien raportointi. (mts. 33.)

Kehittämistutkimus ei varsinaisesti ole oma tutkimusmenetelmänsä, vaan se sisältää joukon erilaisia tutkimusmenetelmiä. Tarpeesta riippuen se voi sisältää kvalitatiivisia tai kvantitatiivisia tutkimusmenetelmiä. Perinteisissä tutkimusmenetelmissä tyydytään esittämään ongelmaan ratkaisu. Kehittämistutkimus taas eroaa perinteisestä laadullisesta ja määrällisestä tutkimuksesta siten, että siinä pyritään myös poistamaan ongelma, eli ottamaan ratkaisu käyttöön. Kuitenkin oikean ratkaisun löytyminen ei ole tae siitä, että ratkaisu toimisi käytännössä. (mts. 39–40.)

3 Ohjelmistotestaamisen perusteet

Tässä luvussa käydään läpi taustatietoa liittyen testaukseen yleisellä tasolla. Luvussa perehdytään eri testauksen menetelmiin ja tasoihin. Lopuksi perehdytään testauksen

suunnitteluun ja dokumentointiin sekä tarkastellaan ylläpitotestausta ja testiautomaatiota.

3.1 Johdatus ohjelmistotestaukseen

Testaus on tärkeä osa ohjelmistokehitystä ja sillä pyritään lisäämään ohjelmistojen laatua sekä vähentämään niissä esiintyviä virheitä. Suurimmalla osalla ihmisistä on ollut kokemuksia ohjelmista, jotka eivät olet toimineet odotetulla tavalla. Ohjelmistojen toimiessa huonosti tulee monenlaisia ongelmia vastaan kuten rahan, ajan tai yrityksen maineen menetys. (Certified Tester Foundation Level Cyllabus 2011, 11.)

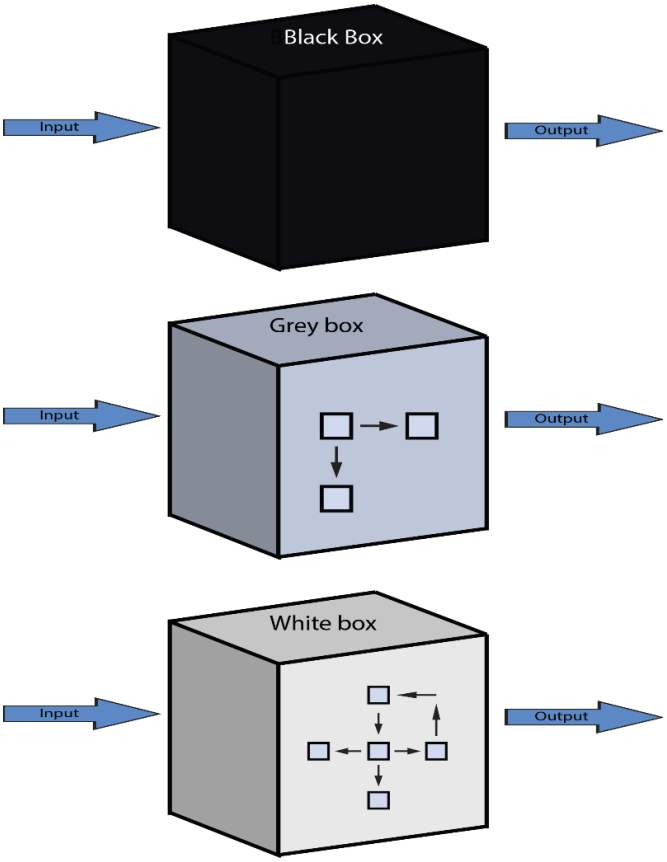
Kehittäjät tekevät sovelluksia tehdessään virheitä, mikä saattaa johtua esimerkiksi ajan puutteesta, monimutkaisesta sovelluksen rakenteesta, uusista teknologioista tai ihmisen erehtyväisestä luonteesta. Nämä tehdyt virheet saattavat näkyä siten, ettei ohjelma toimi odotetulla tavalla. Myös erilaiset ulkoiset tekijät saattavat aiheuttaa virheitä sovelluksen toimintaan. (mts. 11.)

Tarkkaavainen järjestelmän testaus auttaa vähentämään ohjelmistossa esiintyviä virheitä ja auttaa ohjelmistokehittäjiä tuottamaan laadukkaampaa jälkeä. Testaaminen on myös yksi tapa, jolla voidaan mitata sovelluksen laatua. Tästä esimerkkinä virheiden löytyminen, funktionaalisten ja ei-funktionaalisten järjestelmän ominaisuuksien täyttyminen. Järjestelmän laatu parantuukin aina kun virhe löydetään ja kun se saadaan korjatuksi. (mts. 11.)

Testausta toteutetaan moninaisin eri kokoonpanoin. Karkeasti ottaen pienten yksiköiden testaamisesta vastaa ohjelmoija itse ja suurempien kokonaisuuksien kuten järjestelmän testaamisesta vastaa yleensä testaaja tai testaus tiimi. (mts. 24–26.)

3.2 Testausmenetelmät

Testausmenetelmät voidaan jakaa kolmeen eri alakategoriaan (ks. kuvio 1). Nämä kategoriat ovat: Musta-, Valko- ja Harmaalaatikkotestaus. Näissä kolmessa sana laatikko kuvaa järjestelmää ja väri puolestaan kuvaa järjestelmän näkyvyyttä testaajan kannalta.



Kuvio 1. Testauksen laatikkomalli

Musta laatikko -testaus

Musta laatikko -testauksessa ohjelmaa tarkastellaan ikään kuin mustana laatikkona, jonne testaaja ei näe. Tällöin testaaja ei tiedä sovelluksen sisäisestä rakenteesta ja testaajan tulee löytää tilanteet, jossa sovellus ei toimi halutulla tavalla. Siinä testataan usein näkymää, jonka varsinainen käyttäjä sovelluksesta näkee. Etuina tässä testaus metodissa on se, ettei testaajalla tarvitse olla tietämystä kehitystyöstä, ohjelmoinnista tai edes alustasta, minkä päällä sovellusta ajetaan. Musta laatikko -testaus tunnetaan myös termeillä data lähtöinen- tai input/output testaus. (Myers, Sandler, & Badgett 2012, luku 2.)

Musta laatikko -testauksen edut

- Testit suoritetaan käyttäjän näkökulmasta, jolloin epäjohdonmukaisuudet järjestelmän määrittelystä löytyvät.
- Testaajan ei tarvitse osata ohjelmointikieliä tai tekniikoita, joilla itse tuote on toteutettu.
- Testit voidaan tehdä kehittäjistä riippumatta objektiivisesta näkökulmasta.

Musta laatikko -testauksen huonot puolet

- Kaikista mahdollisista syötteistä vain pieniosa pystytään testaamaan.
- Ilman hyvin toteutettua järjestelmän määrittelyä testit on vaikea suunnitella.

(Black Box Testing, n.d.)

Musta laatikko -testausta pystytään hyödyntämään kaikissa testauksen eri työvaiheissa. Tyypillisesti siinä testataan eri käyttötapausten toimivuutta, kuten lomakkeiden lähettämistä ja nappien painamista. Musta laatikko -testausta voidaan käyttää virheellisten ja haitallisten syötteiden testaamiseen; siihen miten järjestelmä menettelee tällaisten syötteiden kohdalla. Musta laatikko -testaukselle tyypilliset testit ovat yleensä niitä testejä, joita automatisoidaan. Tämä johtuu niiden yksinkertaisesta perusluonteesta. (Kasurinen 2013, 65–66.)

Valko laatikko -testaus

Toisin kuin Musta laatikko -testauksessa Valko laatikko -testauksessa testaajan tulee olla tietoinen sovelluksen sisäisestä rakenteesta. Testaajan tulee löytää ohjelmiston poluille oikeat syötteet ja määrittää millainen ulostulon tulee olla. Tämän vuoksi ohjelmointiosaaminen sekä järjestelmän tuntemus ovat erittäin tärkeitä valko laatikko -testauksen onnistumiseksi. Valko laatikko -testaus tunnetaan myös termillä läpinäkyvä laatikko. (White Box Testing, n.d.)

Valko laatikko -testauksen edut

- Testauksen voi aloittaa aikaisessa vaiheessa.
- Testaus on perusteellisempaan, joten sillä saadaan katettua useimmat sovelluksen polut

Valko laatikko -testauksen huonot puolet

- Testiskriptien ylläpito voi olla haastavaa, mikäli ohjelmaan tulee jatkuvasti muutoksia
- Koska testit ovat monimutkaisia, vaativat ne myös tekijältään kattavat taidot niin testaamisen kuin myös ohjelmoinnin osalta.

(White Box Testing, n.d.)

Valko laatikko -testaus on syvällisempää ja kattavampaa kuin musta laatikko -testaus. Tämä johtuu siitä, että testaaja kykenee jäljittämään virheen lähdekooditasolle asti. Välttämättä ei tyydytä siihen, että järjestelmä näyttäisi pintapuolisesti toimivan oikein, vaan pyritään järjestelmän sisäistä toimintaa tutkimalla varmentumaan, ettei tulos ole vain sattumaa. (Kasurinen 2013, 67–68.)

Kasurisen (2013, 68) mukaan ”Lasilaatikkotestaukselle tyypillisimpiä lähestymistapoja ovat kattavuuteen perustuvat mittarit, kuten käytettyjen syötteiden, suoritettujen ohjelmapolkujen tai kokeiltujen komentojen määrä verrattuna kaikkiin vaihtoehtoihin.”

Harmaa laatikko -testaus

Harmaa laatikko -testaus on ikään kuin sekoitus aiemmin esiin tulleita musta laatikko ja valko laatikko testausta. Siinä testaajalla on tiedossa osa testattavan järjestelmän rakenteesta ja toteutuksesta. Tällöin testaaja käyttää sovelluksen sisäistä rakennetta, dokumentteja ja algoritmeja suunnitellessaan testitapauksia, mutta suorittaa testit kuten musta laatikko -testauksessa tehdään. Nimitys harmaa laatikko kuvaa hyvin tilannetta, sillä testaaja näkee vain osan laatikon (järjestelmän) sisällöstä. (Gray Box Testing, n.d.)

Staattinen testaus

Toisin kuin dynaamisessa testaamisessa, joka vaatii ohjelmiston ajamista, staattisen testaaminen sisältää esim. manuaalista koodin tarkastelua ennen kuin ohjelmistoa ajetaan. Näitä kutsutaan tilannekatsauksiksi. Tilannekatsauksessa löydetyt virheet (esim. virheet jotka on määritelty ohjelmiston vaatimuksissa) ovat yleensä paljon nopeampia ja halvempia korjata, kuin ne virheet, jotka havaitaan ohjelmiston ajamisen yhteydessä, eli dynaamisessa testaamisessa. Verrattuna dynaamisen testaamiseen staattisessa testaamisessa löydetään epäonnistumisen syyt (virheet) ennemmin kuin itse epäonnistumiset. (Certified Tester Foundation Level Cyllabus 2011, 32.)

Dynaaminen testaus

Dynaamisessa testaamisessa itse ohjelmaa ajetaan, toisin kuin staattisessa testaamisessa. Dynaamiseen testaamiseen kuuluu tekniikat kuten, yksikkö testaus, integraatio testaus ja järjestelmä testaus. Yksinkertaisesti määriteltynä, dynaaminen testaaminen löytää epäonnistumiset mutta ei varsinaisesti niiden syytä. (mts. 32.)

Graafisen käyttöliittymän testaus

Graafinen käyttöliittymä (engl. graphical user interface) on toinen käyttöliittymä tietokoneohjelmissa tekstipohjaisen komentorivin ohella. Siinä interaktioita ohjelman välillä suoritetaan kuvien ja elementtien avulla. Tästä kohdasta eteenpäin työssä käytetään englanninkielistä lyhennettä GUI. GUI-testaus on testattavan ohjelman testaamista sen graafisen käyttöliittymän kautta. Siinä testataan samoja valikoita, nappuloita ja ikoneita, joita ohjelman varsinaiset käyttäjät käyttävät. (GUI Testing: Complete Guide, n.d.)

Ohjelman käyttöliittymä määrittelee hyvin pitkälle sen aikovatko ihmiset käyttää ohjelmaa vai eivät. Normaalisti käyttäjät havainnoivat ensimmäiseksi ohjelman designia, tuntumaa ja helppokäyttöisyyttä. Mikäli käyttäjien mielestä ohjelma on vaikea käyttöinen ja tuntuma siihen on huono, niin sitä tuskin tullaan käyttämään uudestaan. Tämän vuoksi GUI testaaminen on olennainen osa kokonaistestausta ja on tärkeää, että käyttöliittymä toimii moitteetta. Tärkeitä asioita GUI-testauksessa ovat mm. elementtien koko, syötteitä vastaanottavien kenttien toiminta, ohjelman toiminen GUI:n kautta, virheviestien ja fonttien selkeys, tekstien kohdistus, ulkonäkö ja elementtien responsiivisuus. (mt.)

GUI-testaus voidaan jakaa kolmeen eri menetelmään. Manuaalisessa testauksessa testaaja tarkistaa itse manuaalisesti, että järjestelmä toimii vaatimusmäärittelyiden mukaisesti. Record and Replay (suomeksi nauhoita ja näytä uudelleen) menetelmässä, jokin automaatio ohjelma nauhoittaa testaajan tekemät valinnat ohjelmassa ja myöhemmin testaajan tekemät valinnat voidaan toistaa uudelleen automaatio ohjelman avulla. Viimeinen menetelmä on nimeltään Model Based testing (suomeksi mallipohjainen testaus). Siinä graafisenmallin pohjalta tuotetaan tehokkaita testitapauksia käyttäen järjestelmän vaatimuksia. Se on kehittyvä menetelmä ja sen suurin

etu muihin metodeihin on se, että sillä voidaan määrittää ei toivottuja tiloja mihin GUI voi joutua. (mt.)

GUI testauksen ongelmia ovat puolestaan turhan tiheään muuttuva graafinen käyttöliittymä etenkin, kun muutoksista ei ole saatavilla kunnon dokumentaatiota. Tämä vaikeuttaa testaajien työtä selvästi. GUI testaukseen on saatavilla erilaisia työkaluja kuten QTP, Cucumber ja tässä työssä käytettävä Selenium. Etenkin manuaalinen GUI testaus voi olla aikaa vievä ja itseään toistavaa, joten testiautomaatio on suositeltava ratkaisu GUI testien toteuttamiseksi. (mt.)

Savutestaus

Savutestaus (engl. smoke test) on testausta, jossa varmistetaan, että ohjelmiston perusasiat toimivat. Jos ohjelmiston kuvittelisi olevan jokin sähköllä toimiva laite esimerkiksi televisio ja sen laittaisi seinään, niin savun noustessa laitteesta voisi todeta, että laite on viallinen. Puolestaan ohjelmiston tapauksessa testit voisivat olla tyyliltään seuraavanlaisia: kaatuuko ohjelma käynnistäessä, toimivatko sovelluksen navigointipalkit tai pystyykö sovellukseen kirjautumaan. Savutestejä voidaan käyttää varmentamaan ohjelman yksinkertaiset toiminnot, ennen kuin siirrytään monimutkaisempiin testeihin. Näin saadaan eliminoitua typeriä ja yksinkertaisia virheitä, jotka voivat johtua esimerkiksi puutteellisesta vaatimusmäärittelystä tai testaajien liian tottuneesta testaustyövälineiden käytöstä. (Kasurinen 2013, 72.)

Ylläpitotestaus

Järjestelmä saattaa olla valmistuttuaan käytössä useita vuosia. Tänä aikana järjestelmän käyttöympäristöä saatetaan korjata, muuttaa tai laajentaa. Testausta, joka tapahtuu tällaisessa vaiheessa, kutsutaan ylläpitotestaukseksi (engl. maintenance testing). Ylläpito testaus voidaan jakaa kahteen osa-alueeseen. Ensimmäinen osa-alue sisältää testaamisen, jota suoritetaan, kun järjestelmään tehdään korjaus tai siihen lisätään uusi toiminnallisuus. Toinen osa-alue puolestaan sisältää regressiotestauksen, jolla varmistetaan, että muu järjestelmä toimii muutoksesta huolimatta oikein. (Certified Tester Foundation Level Cyllabus 2011, 30.)

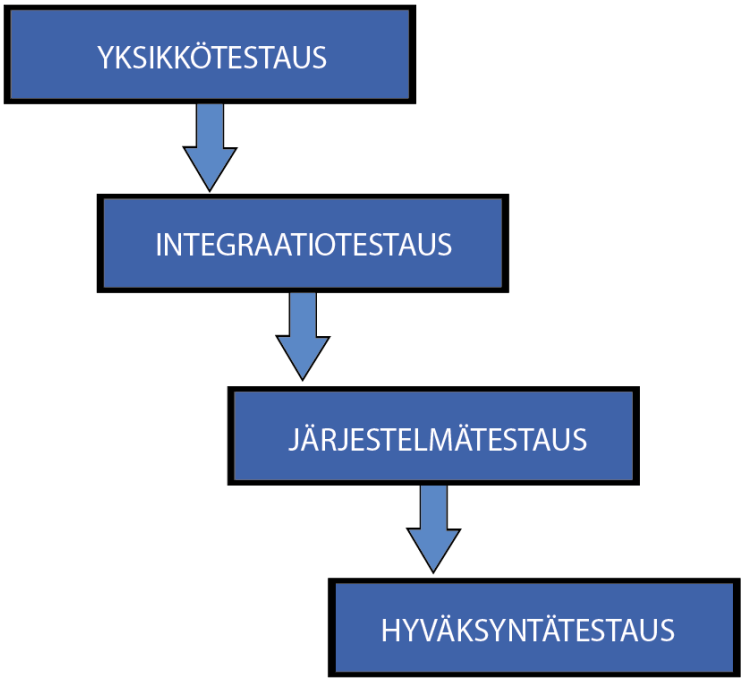
Regressiotestaus on yleistermi, jolla viitattaan uudelleentestaamiseen. Sen käytöstä puhutaan silloin, kun muutoksen jälkeen varmennetaan järjestelmän toimivan oikein. Kasurinen (2013, 69) toteaa kirjassaan

> *Lisäksi regressiotestauksesta voidaan puhua myös silloin, kun kehitettävästä järjestelmästä on saavutettu osatavoite (milestone), ja kyseisestä kehitysversiosta halutaan varmentaa kaikkien toimintojen oikeellisuus; myös niiden jotka oli korjattu aikaisemmassa versiossa.*

Regressiotestaus ei tulisi nähdä minään testauksen tasona tai menetelmänä, vaan ennemminkin työnä, jolla varmistetaan, että muutosten jälkeen ohjelma toimii oikein. Tyypillisesti virhealttiita paikkoja regressiotestaukselle ovat järjestelmän uudet komponentit tai toiminnot, jotka käyttävät näitä komponentteja. Muutettuun osaan tulisi suhtautua aina samalla tavalla, kuin kokonaan uuteen. Regressiotestauksella voidaan varmistaa myös kahden kehityshaaran yhdistämisen tuloksena syntyneen järjestelmän toimivuus. Tällä varmistetaan, että aiemmin korjatut virheet eivät ilmaannu uudestaan yhdistetyssä versiossa. Regressiotestejä toistetaan yleensä usein projektin edetessä, mikä tekee niistä otollisen kohteen testausautomaatiolle. (Kasurinen 2013, 69–70.)

3.3 Testaustasot

Testauksen osiot lajitellaan tyypillisesti neljään eri tasoon: yksikkö-, integraatio-, järjestelmä ja hyväksyttämistestaukseen (ks. kuvio 2). Yleensä testit tehdään tässä järjestyksessä, mutta on olemassa muutamia poikkeuksia, jolloin saatetaan menetellä toisin esimerkiksi tilanteessa, jossa järjestelmä sisältää ulkopuolisten kehittäjien ohjelmistoa. Testaustasojen nimet viittaavat ongelmiin, joita testien avulla pyritään selvittämään. (Software Testing: An ISTQB-BCS Certified Tester Foundation Guide, Third Edition 2015, luku 2.)



Kuvio 2. Testauksentasot

Yksikkötestaus

Yksiköllä (engl. unit) viitataan ohjelman pienimpiin mahdollisiin osiin kuten moduuleihin, luokkiin ja metodeihin. Yksikkö testauksella pyritään varmistamaan, että yksikön koodi vastaa sen vaatimuksia ennen kuin se integroidaan osaksi järjestelmää. Yksikkö testit kirjoittaa yleensä ohjelmoija, joka on kirjoittanut kyseisen yksikön. Näin ollen yksikkö testejä tekevän henkilön tulee päästä järjestelmän ohjelmakoodiin käsiksi. (Software Testing: An ISTQB-BCS Certified Tester Foundation Guide, Third Edition 2015, luku 2.)

Integraatiotestaus

Järjestelmä on integraatio, joka koostuu useista eri yksiköistä. Integraatio testeillä (engl. integration tests) pyritään löytämään virheitä järjestelmän eri osasten interaktioiden välillä. Integraatio testit voivat pohjautua mm. käyttäjäkertomuksiin, järjestelmän arkkitehtuuriin ja suunnitelmiin. Integraatio testeissä testataan tyypillisesti raja-

pintojen toimintaa. Kuitenkin ennen eri osasten yhteen laittamista tulee valita integraatio strategia, jonka mukaan integraatiota lähdetään toteuttamaan. (Software Testing: An ISTQB-BCS Certified Tester Foundation Guide, Third Edition 2015, luku 2.)

Big-Bang Integraatiossa kaikki järjestelmän osaset nivotaan yhteen kerralla. Tällaisen ison järjestelmän testauksessa virheiden löytäminen voi olla vaikeaa ja siksi kyseistä strategia ei usein suositella käytettäväksi. Top-Down (suom. ylhäältä-alas) -integraatiossa järjestelmä kasataan osissa ja ns. ylemmän tason yksiköt testataan ennen alemman tason yksiköitä askel kerrallaan. Tämä vaatii luonnollisesti sen, että alemman tason toteuttamattomia komponentteja joudutaan, jollain tavalla simuloimaan. Down-Top (suom. alhaalta-ylös) integraatio on periaatteeltaan päin vastainen Top-Downiin verrattuna, joten siinä yksikköjen testaus aloitetaan alhaalta ja edetään hierarkiassa ylöspäin. (mt. luvussa 2)

Järjestelmätestaus

Järjestelmätestauksessa (engl. system testing) testataan valmista ja integroitua sovellusta. Järjestelmä testit suoritetaan ympäristössä, joka on mahdollisimman lähellä loppukäyttäjän vastaavaa ympäristöä. Järjestelmä testausta suorittaa yleensä henkilö tai tiimi, joka on erillään kehitysprosessista. Tässä on se hyöty, että toimivuutta voidaan arvioida kirjoitettujen järjestelmän määrittelyiden pohjalta eikä varsinaiseen ohjelmakoodiin perustuen. Tämän vuoksi testaajan ei tarvitse olla tietoinen järjestelmän sisäisestä rakenteesta tai omata vaadittavia ohjelmointitaitoja, joilla järjestelmää on kehitetty. (mt. luvussa 2)

Hyväksymistestaus

Usein järjestelmätestien jälkeen suoritetaan hyväksymistestaukset (engl. acceptance tests). Tarkoitus on tarjota loppukäyttäjälle varmuus siitä, että järjestelmä toimii halutulla tavalla. Tyypillisesti hyväksymistestaus on järjestelmän asiakkaiden tai käyttäjien vastuulla, mutta muutkin projektiryhmän jäsenet voivat olla mukana. (mt. luvussa 2)

Hyväksymistestaus eroaa järjestelmätestauksesta siinä, että erikseen rakennetun testausympäristön sijaan järjestelmää ajetaan sen kohdeympäristössä. Hyväksymistestaus on ikään kuin viimeinen vaihe, jonka jälkeen tuote voidaan todeta olevan onnistunut ja valmistunut. (Kasurinen 2013, 57.)

3.4 Testauksen suunnittelu ja dokumentit

Testaussuunnitelma

Testaussuunnitelma on projektitason dokumentti, jossa määritellään mitä projektissa testataan. Siinä määritellään lisäksi testauksen aikataulu ja käytettävät testausmenetelmät. Yleensä se perustuu organisaation yleisiin linjauksiin testauksesta (organisaaton testausstrategia), mutta se voi myös poiketa näistä käytänteistä. Yleensä testaussuunnitelman laatii projektipäällikkö tai erikseen testaukseen erikoistunut henkilö. Sen rakenne määräytyy projektin tarpeen mukaisesti ja organisaatiolla voi olla useita eri malleja eri tyyppisille projekteille testauksen toteuttamiseksi. Siitä on myös olemassa esimerkiksi kansainvälinen ISO/IEC 29119-testausstandardi. Testaussuunnitelman tekemisen jälkeen laaditaan tyypillisesti testitapaukset. (Kasurinen 2013, 116–118.)

Testitapaukset

Testitapaus on yksittäinen tapahtuma, joka varmistaa jonkin sovelluksen toiminnon. Siinä määritellään siis kaikki vaiheet testitapauksen toteuttamiseksi. Projektin ensimmäiset testitapaukset määritellään arkkitehtuurimallin ja vaatimusmäärittelyyn pohjautuen. Tyypillisesti tämän on tarkoitus varmentaa, että jokin toiminnallisuus pysyy muuttumattomana tai toteutuu. Lisäksi projektiin lisätään testitapauksia, joka kerta, kun havaitaan, että vanhat testitapaukset eivät kata jotain uutta ongelmaa tai kun sovellukseen lisätään uusia ominaisuuksia. (Kasurinen 2013, 118.)

Testitapauksia tulee siis määritelle koko sovelluksen elinkaaren ajan. Kasurisen (2013, 119) mukaan virheiden etsimiseen pätee seuraava sääntö: "Teollisuuden peukalosääntöinä voidaan sanoa, että yksi viidesosa lähdekoodista sisältää neljä viidesosaa virheistä, ja että kymmenen prosenttia löydetyistä virheistä vie yhdeksänkymmentä prosenttia korjauksille varatusta työajasta." Virheiden etsimiseen on olemassa erilaisia ohjeita, joiden avulla ongelma altis koodi voidaan havaita. Kasurisen mukaan

ongelmia aiheuttavat uudet: koodit, ominaisuudet, teknologiat, työntekijät, laitteet tai asiakkaat. Lisäksi muutettu koodi, viime hetken korjaukset ja luonnolliset ihmisten väliset ongelmat aiheuttavat sovellukseen virheitä. (mts. 119–120.)

Testitapaus on kuvaus jostain yksittäisestä tapahtumaketjusta, jonka seurauksena sovellus suorittaa tietyt toimenpiteet. Yksinkertaisimmillaan voisi siis ajatella, että testitapauksen tulee kertoa mitä testitapauksessa tehdään ja mikä on oletettu tulos. Tämä ei kuitenkaan aina riitä vaan on olemassa liuta muita oleellisia seikkoja, joita tulee kirjata ylös. (mts. 120.)

Kasurisen (2013, 120–121) mukaan on olemassa useita seikkoja, joita testitapauksessa voidaan määritellä:

- kuka testitapauksen on tehnyt?
- mihin se vaikuttaa?
- mistä saa lisätietoa?
- testitapauksen konteksti
- tavoite
- syötteet
- lopputulos
- käytettävä ympäristö
- erikoisehdot
- riippuvuudet
- laatuehdot
- rajoitteet.

3.5 Testiautomaatio

Testiautomaatiossa pyritään automatisointityövälineillä tekemään testauksesta automaattista. Tarkoituksena on siis asettaa usein toistuvia testejä erilliselle laitteelle nopeaa tarkastusta varten. Näin saadaan vapautettua testaajia muihin tehtäviin. Testit voidaan asettaa esimerkiksi yön aikana tapahtuviksi, jolloin kehittäjät voivat seuraavana työpäivänä tarkistaa testien tulokset ja korjata mahdolliset virheet. (Kasurinen 2013, 76.)

Kasurisen (2013, 76) mukaan "Testausautomaatiolle kaikille otollisimpia kohteita ovat esimerkiksi moduulien rajapinnat tai yksittäisten moduulien yksikkötestauksessa tehtävät tarkastukset." Lisäksi käyttöliittymien tarkastukset on ollut toinen otollinen käyttökohde testausautomaatiolle. (mts. 76.)

Testausautomaatien ajatellaan usein korvaavan käsin testauksen, vaikka asia tulisi nähdä ennemmin niin, että se on vain käsin testausta täydentävää toimintaa. Useat väärät mielikuvat automaattisesta testauksesta ovat aiheuttaneet monenlaisia ongelmia ohjelmistokehityksessä. Testausautomaatio vaatii ylläpitoa ja resursseja kehitystiimiltä siinä missä perinteinen käsin testaus. Testiautomaation käyttö asettaa tiettyjä vaatimuksia projektille ja aivan kaikissa projekteissa sen käyttö ei välttämättä ole perusteltua. (mts. 76–77.)

Mikäli yksittäistä testiä suoritetaan n. 4-20 kertaa projektin kuluessa on syytä harkita testausautomaation käyttöönottoa. Testausautomaatiolla tulee ensisijaisesti varmistaa, että aiemmin toimineet osat toimivat, kun järjestelmään lisätään uusia ominaisuuksia. (mts. 77–78.) Kasurisen kirjassa (2013, 78) todetaan osuvasti: ”automaation avulla estetään ehjiä moduuleja rikkoutumasta ja käsin testauksella tutkitaan uusia tapoja rikkoa toiminnallisuuksia.” Automaattisissa testeissä itse testien tekeminen on kallista, mutta niiden ajaminen ei varsinaisesti tuota lisäkustannuksia. Kiinnostuneisuus testausautomaatio kohtaan on teollisuuden puolesta ollut hidasta, mutta kuitenkin nousujohteista. Oikeastaan ongelmana testausautomaation leviämisessä on ollut sen käyttökohteiden väärinymmärrykset ja siitä seuranneet epärealistiset odotukset ja ongelmat. (mts. 78–79.)

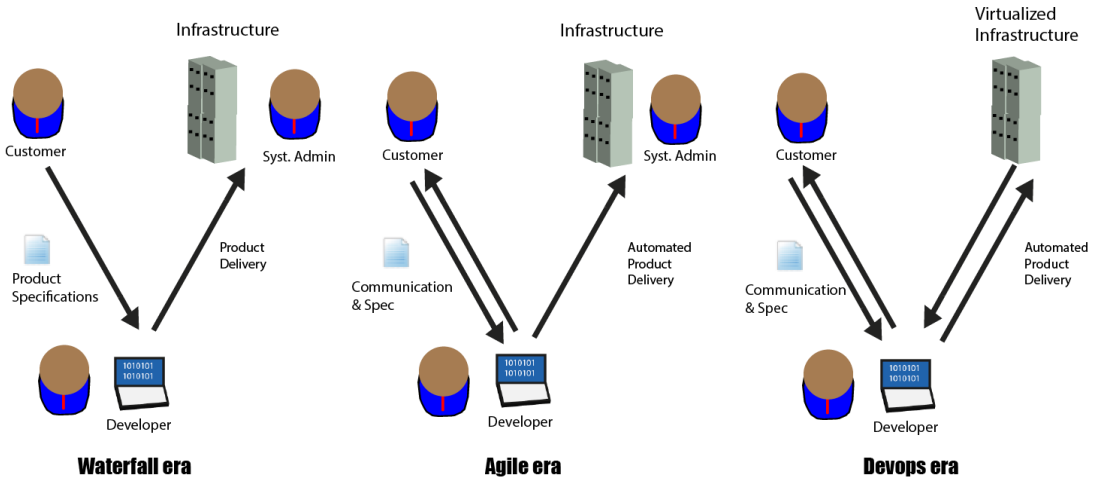
4 DevOps ja Jatkuva integraatio

Tässä luvussa käydään läpi DevOps sovelluskehitysmenetelmää ja siihen sisältyvää jatkuvaa integraatiota. Syvemmin pureudutaan jatkuvan integraation hyötyihin ja haittoihin ja siihen liittyviin käytänteisiin. Lisäksi perehdytään jatkuvan integraation esiasteeseen ns. yölliseen käännökseen ja tutkimuksessa käytettävään automaatiotyökaluun Jenkinsiin.

4.1 DevOps

Ensimmäinen suuren suosion saanut sovelluskehitys menetelmä oli vesiputousmalli. Se perustuu tarkkoihin ennalta määriteltyihin vaatimuksiin ja näiden vaatimusten sopimuksen mukaiseen joustamattomaan noudattamiseen. Vesiputousmalli vastasi

huonosti mahdollisiin muutoksiin projektissa ja viestiminen asiakkaan kanssa oli yksisuuntaista. (Eficode Quick Guide: DevOps, 6–7)



Kuvio 3. Sovelluskehityksen aikakaudet

Ketterät kehitysmenetelmät ratkaisivat monia vesiputous mallin ongelmia. Ne tekivät tuotekehityksestä joustavaa jakamalla projektin työkuorman sykleihin. Ne painottivat myös asiakasviestinnän tärkeyttä. Lisäksi niiden avulla päästiin eroon monista muista vesiputousmallin tuottamista ongelmista. Ketterät kehitysmenetelmät eivät kuitenkaan ottaneet huomioon työskentely ympäristöä ja kehityksessä käytettäviä työkaluja. Tämän vuoksi kehitysprosessiin muodostuu pullonkauloja, jotka hidastavat sitä. Näiden asioiden korjaamiseksi muodostui ns. kolmannen sukupolven kehitysmenetelmä DevOps (mts. 7.)

DevOps on ohjelmistokehityksen menetelmä, jolla ei ole virallista määritelmää. Se voidaan kuitenkin ajatella olevan laaja joukko erilaisia toimintoja ja työkaluja, jotka on kehitetty parantamaan ketterää kehitysmenetelmää. DevOps perustuu kehitysprosessin automatisointiin, jolloin sovelluskehittäjät voivat keskittyä itse kehitykseen. Se muodostuu sanoista kehitys (engl. development) ja tuotanto (engl. operations) (mts. 2.) Organisaatio, joka käyttää DevOps kehitysmenetelmää julkaisee ja testaa

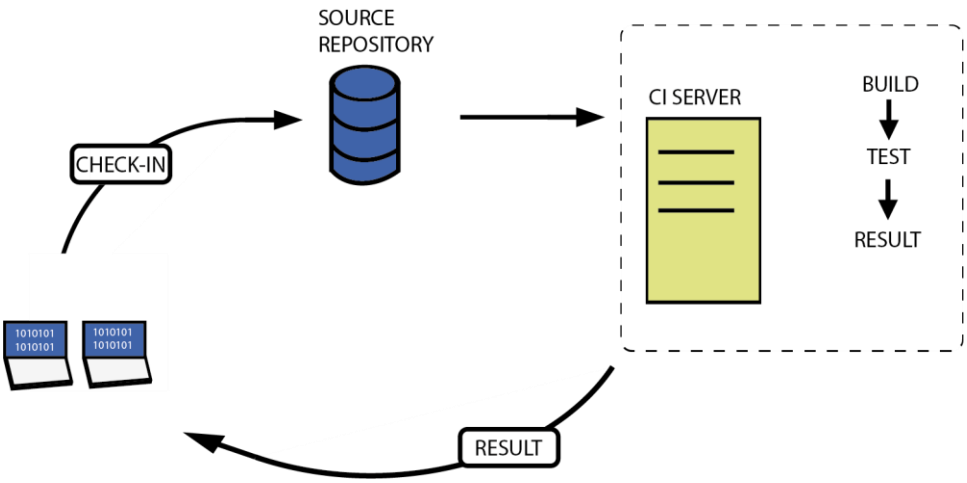
ohjelmistoa jatkuvasti ja automatisoidusti. Kehitystyö tehdään yleensä virtuaalisessa ympäristössä, joka vähentää virheitä ja rahanmenoa infrastruktuurin hallintaan ja kehitystyöhön. Tekniset ratkaisut ovat tärkeässä roolissa DevOps:ssa, kuten esimerkiksi pilvipalvelut ja manuaalisten prosessien automatisointi (mts. 7.)

DevOps tarjoaa useita ratkaisuja ketterien kehitys menetelmien ongelmiin. DevOps mahdollistaa uusien ominaisuuksien automaattisen testaamisen ja julkaisun heti kun ne ovat valmistuneet. Näin säästetään rahaa ja pysytään paremmin aikataulussa. Ketterissä menetelmissä sovelluskomponentit eivät ole aina yhteensopivia, mutta DevOps:ssa avoimet rajapinnat ja automatisointi mahdollistavat projektin jakamisen erillisiin mutta yhteensopiviin osiin. Ketterissä menetelmissä tuotteen laatua ei usein varmenneta riittävän hyvin ennen julkaisua ja uudet ominaisuudet saattavat rikkoa vanhoja. Puolestaan DevOps:ssa automaattinen testaus vähentää manuaalisen testauksen tarvetta ja varmistaa sen, että vanhat ominaisuudet eivät hajoa uusien tullessa julkaisuun. DevOps myös lisää yritysten sisällä eri tahojen kommunikaatiota ja selkeyttä vastuita näiden välillä (mts. 7.)

DevOps ei kuitenkaan ratkaise kaikkia ohjelmistokehityksen ongelmia eikä se poista manuaalista työtä kokonaan. Se kuitenkin vähentää itseään toistavia työtehtäviä ja sovelluskehittäjät voivat keskittyä itse kehitystyöhön sekä laadukkaan ja käyttäjäystävällisen tuotteen tekemiseen (mts. 8.)

4.2 Jatkuva integraatio

Ohjelmistokehityksessä jatkuvalla integraatiolla tarkoitetaan ohjelmoijien töiden tai dokumenttien yhdistämistä yhteiseen säilytyspaikkaan. Ohjelmoijat integroivat pieniä osia ohjelmistosta jatkuvasti projektiin (ks. kuvio 4). Jatkuva integraatio paljastaa mahdolliset viat aikaisessa kehitysvaiheessa ja pienemmässä skaalassa ennen kuin ohjelmisto on valmis, tehden näin vikojen korjaamisesta helpompaa. Jos viat saadaan korjattua ennen ohjelmiston tuotanto versiota, säästetään kustannuksissa. (Berg 2012.)



Kuvio 4. Jatkuvan integration sykli

4.2.1 Hyödyt ja Haitat

Jatkuvan integraation kulmakivenä toimii ajatus: integroi varhain ja usein. Tämän saavuttamiseksi asetetaan usein integraatioserveri, joka on vastuussa jatkuvien tuontiversioiden kokoamisesta ja testaamisesta. On olemassa useita kaupallisia ja ilmaisia ohjelmia, joilla tämä voidaan saavuttaa. (Cauldwell 2008, luku 3.)

Kaksi suurinta tavoitetta jatkuvassa integraatiossa ovat Cauldwellin (2008, luku 3) mukaan

- Varmistua siitä, että ohjelmistosta on aina testattava versio, joka heijastaa uusinta koodia.
- Varoittaa kehittäjiä integraatio ongelmista mahdollisimman nopeasti.

Jatkuvalla integraatiolla pyritään vähentämään kehitykseen liittyviä riskejä. Aiemmin integraatio vaiheen kesto on ollut vaikea arvioida ja se on näkynyt viivästyneissä projekteissa. Integraatio vaihe on ollut aiemmin ikään kuin sokea piste, jonka jatkuva integraatio on saanut poistettua. Nykyisin ollaan tietoisia sovelluksen tilasta, mitkä sovelluksen osat ovat toimivia ja mitkä eivät. (mt. luvussa 3)

Jatkuva integraatio ei kuitenkaan poista ohjelmistosta virheitä, mutta se tekee niiden löytämisestä huomattavasti helpompaa. Yleensä virhe havaitaan juuri sellaisesta kohdasta mitä ohjelmoija on hetki sitten työstänyt. Tällöin ohjelmoijalla on tuoreessa muistissa kohta mitä hän on työstänyt ja virheen selvittäminen on täten nopeampaa. Virheillä ohjelmissa on usein taipumus kasaantua ja useita virheitä on vaikeampi korjata, kuin yksittäisiä. Jatkuva integraatio poistaa tätä ongelmaa, sillä ohjelmistoon lisätään ominaisuuksia pala kerrallaan useasti, jolloin vika on helpompi hahmottaa. Usein jatkuvaa integraatiota käytävillä projekteilla on huomattavasti vähemmän virheitä tuotannossa ja itse prosessissa. Kuitenkin saadut hyödyt ovat sidoksissa siihen, miten hyvin testit on toteutettu. (Fowler 2006.)

Jatkuvan integraation avulla käyttäjät saavat nopeammin uusia ominaisuuksia käyttöönsä ja kykenevät antamaan kehittäjien suuntaan palautetta uusista ominaisuuksista nopeammin. Näin kehittäjien ja asiakkaiden välillä oleva seinä ikään kuin poistuu, mikä on yksi ohjelmistokehityksen suurimmista ongelmista. (Fowler 2006.)

Kaikki eivät kuitenkaan käytä jatkuvan integraation menetelmiä niiden lukuisista eduista huolimatta. Tämä johtuu yksinkertaisesti siitä, että se on vaikeaa. Palvelinpuolen asetusten kuntoon laitto, taito kirjoittaa hyviä testejä ja jatkuvan integraation metodien hallitseminen ovat muutamia asioista, joita vaaditaan jatkuvan integraation toimimiseksi. Usein kuitenkin vaikeinta on saada idea myydyksi omalle tiimille, joka on tottunut toimimaan eri tavalla. Lisäksi päivittäinen kommitointi ja muiden ihmisten koodiin muuttaminen voivat olla tietyntyyppisille ihmisille vaikeasti omaksuttavia. (Cauldwell 2008, luku 3.)

4.2.2 Käytänteet

Tiettyjä ohjeistuksia tarvitaan, jotta jatkuva integraatio saadaan toimimaan sulavasti. Fowlerin (2006) kirjoittamassa artikkelissa kuvataan hyödyllisiä käytänteitä jatkuvan integraation toteuttamiseksi.

Ylläpidä yhtenäistä lähdevarastoa

Ohjelmat koostuvat useista organisoiduista tiedostoista, joista lopuksi kasataan kokonainen toimiva ohjelma. Näiden tiedostojen seuranta vaatii suurta ponnistelua etenkin, kun sovellusta on kehittämässä useampi henkilö. Tämän vuoksi on kehitetty

useita eri työkaluja, joilla näitä tiedostoja pystytään seuraamaan tehokkaasti. Tällaisia sovelluksia kutsutaan versionhallintaohjelmiksi/järjestelmiksi.

Versionhallintajärjestelmään tulee laittaa kaikki sellaiset tiedostot, joita vaaditaan sovelluksen ajamiseksi. Käytännössä tämä tarkoittaa sitä, että projektin ulkopuolisen henkilön tulisi saada yhdellä latauksella versionhallintajärjestelmästä itselleen tiedostot, joiden avulla hän kykenisi ajamaan sovellusta omalla tietokoneellaan. (Fowler 2006.)

Automatisoi käännöksen toteutus ja tee prosessista automaattisesti testaava

Kun lähdetiedosto muutetaan ajattavaksi järjestelmäksi, muodostuu käännös (engl. build). Perinteisesti tämä tarkoittaa lähdetiedostojen kääntämistä, linkitystä ja muita asioita, joilla ohjelma saadaan suoritetuksi. Prosessi on luonteeltaan sellainen, että se on mahdollista automatisoida ja se on kannattavaa, sillä näin säästetään kallisarvoista aikaa kehityksessä. Yleinen virhe on, ettei kaikkea ohjelman toiminnan kannalta oleellista oteta mukaan automaattiseen käännökseen. (Fowler 2006.)

Ohjelma saattaa toimia, mutta silloin tällöin sekaan pääsee myös virheitä. Tämän vuoksi automaattista käännöstä ajettaessa on hyvä suorittaa myös automaattisia testejä. Testit eivät kuitenkaan ole koskaan täydellisiä, mutta niillä voidaan löytää suurin osa virheistä. Prosessin tulisi toimia siten, että mikäli automaattisissa testeissä tapahtuu virhe niin käännöstä ei toteuteta. (Fowler 2006.)

Jokainen kommitoi päähaaraan päivittäin

Integraatio on pohjimmiltaan kommunikointi, jonka avulla kehittäjät voivat kertoa toisilleen, mitä muutoksia he ovat tehneet. Usein tapahtuva kommunikointi antaa kehittäjille tiedon, milloin muutoksia tapahtuu. Kun kehittäjät siirtävät päivittäin tekemänsä muutokset versionhallintaan, ovat he tietoisi muiden tekemistä muutoksista ohjelmaan. Tällöin ristiriita kahden kehittäjän tuotoksien välillä saadaan korjatuksi suhteellisen pienellä työmäärällä. Tilanne olisi kuitenkin toinen, mikäli muutoksia siirrettäisiin versionhallintaan esimerkiksi kerran viikossa; virheet olisivat paisuneet suuriksi ja niiden korjaaminen veisi huomattavan määrän aikaa. Käytäntö, jossa siirre-

tään muutokset päivittäin versionhallintajärjestelmään kannustaa kehittäjät jakamaan työnsä pieniin paloihin, jolloin työn edistymistä on myös helpompi seurata. (Fowler 2006.)

Jokainen päähaaraan lisätty kommitti tulee ajaa integraatiokoneen kautta.

Teoriassa käännös pysyy aina puhtaana, mikäli kehittäjät noudattavat yllämainittua joka päivä tapahtuvaa muutosten siirtämistä versionhallintaan. Kuitenkin kehittäjien koneille saattaa olla eroavaisuuksia tai yksinkertaisesti ollaan laiskoja päivittämään versiota ennen oman version siirtämistä palvelimelle. Tämän vuoksi kehittäjän tulee varmistaa, että säännöllisiä käännöksiä toteutetaan vain, mikäli kehittäjän integraatio versio läpäisee testit ja ei aiheuta ristiriitoja versionhallintajärjestelmässä. Tähän on olemassa kaksi tapaa manuaalinen käännös ja integraatioserverin käyttäminen. (Fowler 2006.)

Käännöksen virheet tulee korjata välittömästi

Jatkuvien käännösten tekemisessä tärkeintä on saada virheet korjattua välittömästi niiden ilmaannuttua. Jatkuvan integraation perus periaatteena on kehittäjien tekemän koodin toteutus vakaalle koodi pohjalle. Mikäli versionhallinnan päähaarassa havaitaan virhe, on erittäin tärkeää, että se korjataan mahdollisimman nopeasti. Usein tämä onnistuu helpoiten menemällä versionhallintaohjelmassa yksi versio taaksepäin eli ottamalla käyttöön viimeisin vakaa version ohjelmasta. Toisaalta, jos virhe on ilmiselvä, voidaan se selvittää nopeasti kehittäjän työasemalla debuggerityökalun avulla. (Fowler 2006.)

Käännöksen rakentamisen tulee olla nopeaa

Koko jatkuva integraatio perustuu välittömään palautteeseen. Yleisesti ottaen 10 minuutin odotus käännöksen valmistumisessa on hyväksyttävä aika, mutta mitä nopeammin käännös valmistuu sitä parempi. Toisaalta mikäli käännöksessä tuntuu kestävän liian kauan, niin yksi mahdollisuus on pystyttää käännös linjasto (engl. build pipeline). Käytännössä tarkoitus on tehdä useita käännöksiä peräkkäin. Asiaa voi hahmottaa yksinkertaisesti siten, että kuvitellaan tilanne, jossa käännöksen teko on kaksivaiheinen. Kehitystiimistä joku tekee kommitin ja kommitti kulkee ensimmäiselle käännöskoneelle. Kyseinen kone tekee nopeat yksinkertaiset testit pysyen n. 10 minuutin

aikamääreessä kiinni. Toinen kone tekee puolestaan laajemmat ja aikaa vievät testit. Tämä mahdollistaa sen, että kehittäjät voivat alkaa rakentaa ensimmäisen käännöskoneen tekemän käännöksen päälle uusia ominaisuuksia heti kun se on valmistunut. Toisaalta mikäli toisen käännöskoneen laajat testit eivät mene läpi täytyy voimavaroja siirtää kyseisen virheen korjaamiseksi. Tätä peräkkäin käännös tapaa voidaan soveltaa myös useilla muilla konemäärillä, mikäli se koetaan tarpeelliseksi. (Fowler 2006.)

Testit tulee suorittaa tuotantoympäristöä vastaavassa ympäristössä

Testi ympäristö tulisi aina asettaa tuotanto ympäristöä vastaavalla tavalla esimerkiksi tietokantaohjelmistolla tulisi olla sama versio, käyttöjärjestelmän ja käytettyjen kirjastojen tulisi olla vastaavat jne. Näin vältetään ympäristöstä aiheutuvia virheitä. Johonkin täytyy kuitenkin asettaa raja, miten tarkkaan kohdeympäristöä voidaan simuloida. Toisaalta, mikäli käännösten teko on jaettu useille koneille, voidaan ensimmäisen koneen kohdalla, joka tekee nopeat testit, käyttää yksinkertaista ympäristöä ja vasta toisen käännöskoneen kohdalla tehdä testit tuotantoympäristössä. Näin saadaan pidettyä ensimmäinen käännös nopeana. (Fowler 2006.)

Kaikille tulee tarjota pääsy käyttämään ohjelman uusinta versiota

Ohjelmistokehityksessä yksi vaikeimmista asioista on varmistua siitä, että tehdään oikeanlaista ohjelmaa. Ihmisten on paljon helpompi kertoa, mitä he sovelluksesta haluavat, kun he näkevät edes osin toimivan version. Tällöin he voivat kertoa, mitä parannettavaa sovelluksessa on. Ketterät kehitysmenetelmät käyttävät juuri tätä ihmisen käyttäytymisen piirrettä hyväkseen kehityksessä. Kaikkien projektissa olevien tulisi siis saada pääsy kokeilemaan ja tarkastelemaan ohjelmaa. (Fowler 2006.)

Kaikkien tulee nähdä mitä tapahtuu

Kommunikaation ollessa jatkuvan integraation kulmakivi on hyvin tärkeää, että projektiin kuuluvat ihmiset näkevät päähaaran käännösten tilan. Usein jatkuvan integraation sovelluksissa on toimintoja kuten päivittyvät verkkosivut, jonka kautta voi käydä tarkastelemassa mm. käännösten rakentamisen tilaa, ja erilaisia lokeja. Tällaisten sivujen pitäminen on myös hyvä myös projektiin kuuluvien toisaalla työskentelevien tahojen ja etenkin asiakkaan kannalta. (Fowler 2006)

Automaattinen käyttöönotto

Jatkuvan integraation toteuttamiseen tarvitsee useita ympäristöjä kuten, kommitointi testeille ja toissijaisille testeille. Koska näiden ympäristöjen välillä liikutellaan useasti dataa, on näiden ympäristöjen toiminta ja tiedonsiirto syytä automatisoida. Näin ollen tulee tehdä skriptit, joilla sovelluksen saa otettua käyttöön eri ympäristöihin helposti. Kuitenkin automaattisessa käyttöönotossa on myös syytä olla mekanismi, millä testit läpäissyt virheellinen sovellus saadaan nopeasti vaihdettua viimeisimpään toimivaan versioon, mikäli tuotantoon pääsee livahtamaan virheellinen versio. Yksi tapa on myös tarjota rajatulle käyttäjäjoukolle kokeilu tai beeta versiota ennen kuin siirtää sovelluksen käytettäväksi suuremmalle ihmismäärälle. (Fowler 2006.)

4.2.3 Jatkuva integraatio vs. yöllinen käännös

Nokialla oli aikanaan käytössä metodi nimeltään yöllinen käännös. Sitä voidaan pitää jatkuvan integraation esiasteena. Siinä automatisoitu järjestelmä hakee, joka yö päivänaikana tehdyt muutokset versionhallintajärjestelmästä ja tekee niiden pohjalta käännöksen. Idea on hyvin samankaltainen kuin jatkuvassa integraatiossa, mutta yönaikainen muutostenmäärä on huomattavasti suurempi kuin useasti päivässä tapahtuvassa jatkuvassa integraatiossa. Tämän vuoksi yöllisten käännösten virheiden korjaus on ollut haastavaa. Nokia siirtyi myöhemmin käyttämään jatkuvan integraation periaatteiden mukaisia käytänteitä ja näin saatiin huomattavasti vähennettyä aikaa uusien sovellusten julkaisussa. (Saurabh. 2016a.)

4.2.4 Jenkins

Jenkins on avoimen lähdekoodin automaatiotyökalu, joka on tarkoitettu käytettäväksi jatkuvaan integraatioon. Se on Java pohjainen ja sen avulla voi jatkuvasti testata ja tehdä ohjelmasta käännöksiä. Jenkinsiin on saatavilla useita eri liitännäisiä, joiden avulla siihen voi lisätä toiminnollisuuksia, esimerkiksi liitännäiset versionhallintajärjestelmään ja jatkuvaan testaukseen. (Saurabh. 2016a.)

Saurabh (2016a) mukaan Jenkinsin käytössä on seuraavia etuja:

- Avoimen lähdekoodin ohjelma hyvällä yhteisön tuella.
- Se on helppo asentaa.

- Sille on tarjolla yli 1000 erilaista liitännäistä ja tarvittaessa sellaisen voi ohjelmoida itse.
- Se on ilmainen käyttää.
- Java-pohjaisuus tekee siitä siirrettävän tärkeimmille alustoille.

Lisäksi Jenkinsillä on maailmanlaajuisesti yli miljoona käyttäjää ja yli 147 000 aktiivista asennusta. Jenkinsille on tarjolla yli 1000 liitännäistä, joiden avulla on mahdollista yhdistää tärkeimmät kehitys-, testaus- ja käyttöönottotyökalut. (Saurabh. 2016a.)

Asennus

Jenkinsin kotisivut tarjoavat suurimmalle osalle käyttöjärjestelmistä natiivin asennuspaketin ladattavaksi Jenkinsin kotisivulta. Jenkinsin voi ladata myös WAR tiedostomuotona, jolloin asennuspaketti käynnistetään komentorivin kautta. Asennuksen jälkeen Jenkinsin tulisi näkyä selaimessa osoitteessa: http://localhost:8080/ (Vogel 2017.)

Lisäksi Jenkinsistä on tarjolla LTS-versio (Long-Term Support) eli versio, jota päivitetään harvemmin ja johon lisätään vain tärkeimmät päivitykset. Tämä on hyvä käyttäjille, jotka haluavat Jenkinsin olevan vakaa esimerkiksi tilanteessa, jossa yritys ylläpitää omaa kustomoitua Jenkinsin versiota. (LTS Release Line, n.d.)

Konfigurointi

Jenkinsin asetuksia voidaan muokata sen verkkokäyttöliittymästä. Suositeltavaa on aktivoida turvallisuusasetukset ja luoda ainakin yksi anonyymikäyttäjä lukuoikeuksilla. Näin uudet käyttäjät eivät pääse muuttamaan mitään Jenkinsissä. Lisäksi tilanteessa, jossa on tarve ottaa yhteyttä yksityiseen versionhallintajärjestelmään esimerkiksi Gittiin, on luotava SSH-avain. Tämä onnistuu komennolla sudo -u jenkins ssh-keygen. Komennolla saatu julkinen avain tulee laittaa käytettävään palveluun eli tässä tapauksessa Githubiin. (Vogel 2017.)

Jenkins käännöstyöt

Jenkinsin käännös työ (engl. build job) on erityinen tapa kääntää, testata, paketoida, lähettää tai tehdä jotain projektille. Jenkins tukee useita eri tyyppisiä käännöstöitä. Smart (2011, 81–82) määrittelee oppaassaan seuraavia eri käännöstöitä.

Freestyle sovellus projektit ovat joustavia ja yleistarkoituksellisia ja niillä voidaan tehdä monenlaisia tehtäviä.

Maven käännöksenhallintajärjestelmää käyttäviä projekteja voidaan käyttää melko suoraan Jenkinsissä. Apache Maven on Java kehityksessä käytettävä käännöksenhallintatyökalu. Se käyttää POM:a (engl. project object model) projektin tietojen tallennukseen. (What is Maven? 2017) Tyypillisesti se on pom.xml niminen tiedosto projektissa.

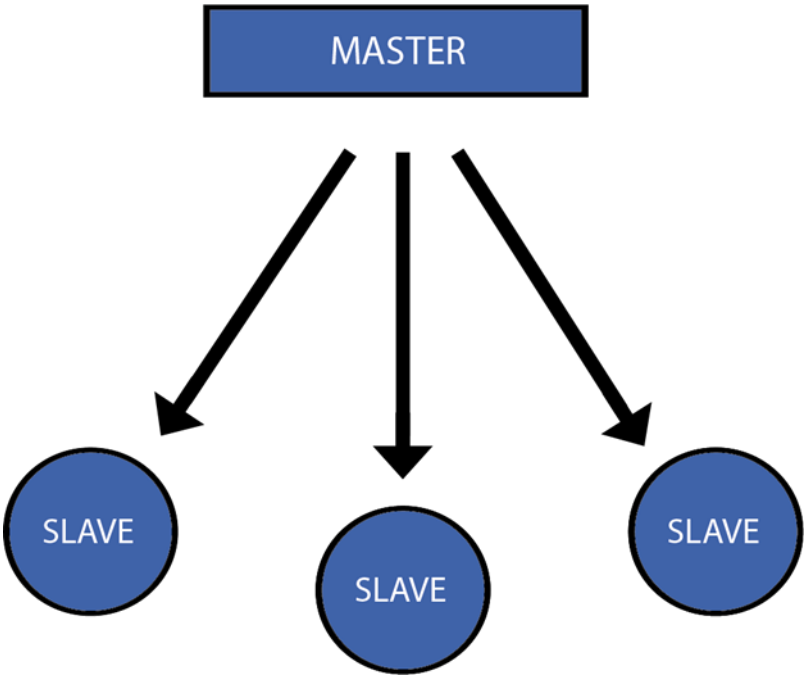
Moni konfigurointi projektit (engl. multi-configuration projects) ovat Jenkinsissä erittäin tehokas ominaisuus. Ominaisuus on kätevä esimerkiksi silloin, kun projektia käännetään usealle eri alustalle.

Liitännäiset

Jenkinsin toimintaa on mahdollisuus kasvattaa erilaisten liitännäisten avulla. Liitännäisillä on mahdollisuus tehdä käytännössä mitä vaan aina versionhallinnastajärjestelmistä ilmoitus työkaluihin. Liitännäisiä on mahdollisuus asentaa, poistaa ja päivittää ”manage plugin” näkymän kautta. (Smart 2011, 67.) Mikäli jotain liitännäistä ei löydy on käyttäjän myös mahdollista kirjoittaa omia liitännäisiä. Jenkinsin www-sivut tarjoavat kattavaa opastusta omien liitännäisten kehittämiseen.

Skaalattavuus

Joskus suuremmissa ja raskaammissa projekteissa on tarve jakaa käännösten suorittamista useammilla eri koneille. Jenkins käyttää mestari-orja (engl. master-slave) arkkitehtuuria jaettujen käännösten hallintaan (ks. kuvio 5). Tällöin ”pää”-kone on mestari, jonka tehtävänä on hallita aikataulutusta, lähettää käännöksiä orjille suoritettavaksi, monitoroida orjien toimintaa ja tallentaa ja esittää käännösten tuloksia. Tässä mallissa, jopa mestari voi suorittaa käännös töitä suoraan. Orjien on taas puolestaan toimittava niin kuin niitä käsketään, mikä tarkoittaa käytännössä käännösten suorittamista. Orjat voivat toimia eri käyttöjärjestelmillä ja kommunikointi orjien ja mestarien välillä tapahtuu TCP/IP protokollan avulla. (Saurabh. 2016b.)



Kuvio 5. Master/Slave-arkkitehtuuri

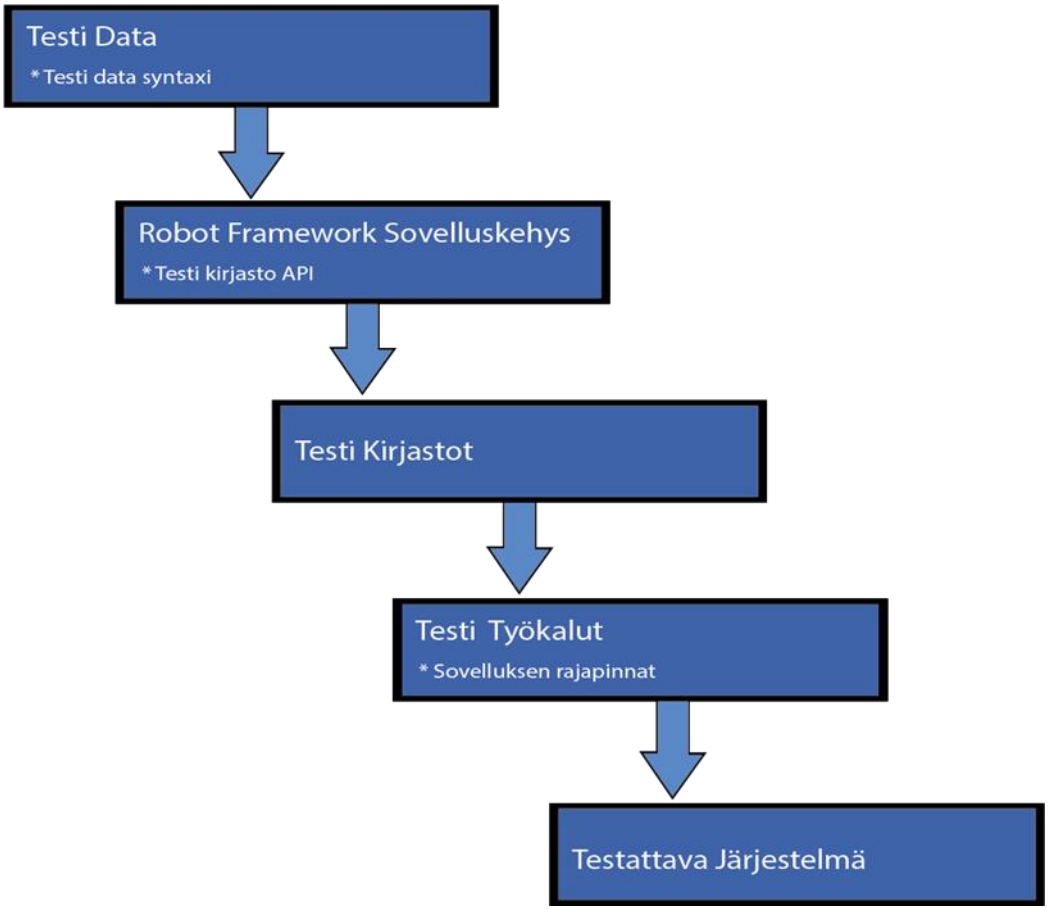
5 Robot Framework

Tässä luvussa tutustutaan työssä käytettävään testiautomaatiokehykseen Robot Frameworkkiin. Lisäksi tutustutaan testien kirjoittamiseen, suorittamiseen ja testi tuloksiin. Lopuksi käydään läpi työn kannalta oleellisia testikirjastoja ja käytänteitä, joiden avulla koodista tehdään selkeää ja ylläpidettävää.

5.1 Robot Frameworkin perusteet

Robot Framework on testiautomaatiokehys hyväksymistestaukseen ja hyväksyntä testi pohjaiseen kehitykseen (ATDD). Se käyttää avainsana pohjaista menettelytapaa ja tabuloitavaa testidatasyntaksia, joka on helppo oppia ja käyttää. Avainsana syntaksia käyttäen luodaan testitapaukset ja sovelluskehykseen voidaan lisätä useita ulkoisia kirjastoja helpottamaan toimintoja, kuten Selenium2 kirjasto, jota käytetään www-sovellusten ja sivujen testaamiseen (ks. kuvio 7). Itse sovelluskehys on avoimen lähdekoodin sovellus, joka perustuu Python ohjelmointi kieleen. Sitä voidaan myös

ajaa Jython:in kautta, joka pohjautuu Java kieleen ja Iron Python:illa, joka on Pythonin C#-kielinen toteutus. Se on julkaistu Apache 2.0 lisenssin alle ja sen kehitystä tukee Nokia Networks. (Robot Framework User Guide Version 3.0.2.)



Kuvio 6. Robot Frameworkin arkkitehtuuri

5.2 Asentaminen

Robot Framework vaatii toimiakseen Pythonin version 2 tai 3. Käytettäessä Jythonia tarvitaan Java-alusta ja puolestaan Ironpython vaatii .NET-alustan. Tähän opinnäytetyöhön on valikoitunut Pythonin versio 2.7.13. Pythonin standardi paketti manageri on nimeltään pip. Sen avulla Robot Frameworkin asentaminen sujuu suoraviivaisesti

yhdellä komennolla: pip install robotframework. (Robot Framework User Guide Version 3.0.2.)

5.3 Testien kirjoittaminen

Testien ohjelmointi perustuu Robot Frameworkin omaan avainsanapohjaiseen kieleen. Avainsanoja voidaan tehdä yhdistelemällä olemassa olevia avainsanoja tai kirjoittaa itse Pythonilla tekemällä mukautettuja kirjastoja. Käytännössä testejä voidaan kirjoittaa lähes millä tahansa tekstieditorilla, mutta integroidut kehitysympäristöt tarjoavat paljon kehitystä helpottavia toimia, kuten esimerkiksi syntaksin tarkastuksen, projektissa navigoinnin ja debukkauksen. Suoritettavia testikokonaisuuksia kutsutaan Robot Frameworkissa testidataksi. (Robot Framework User Guide Version 3.0.2.)

Testitapaus

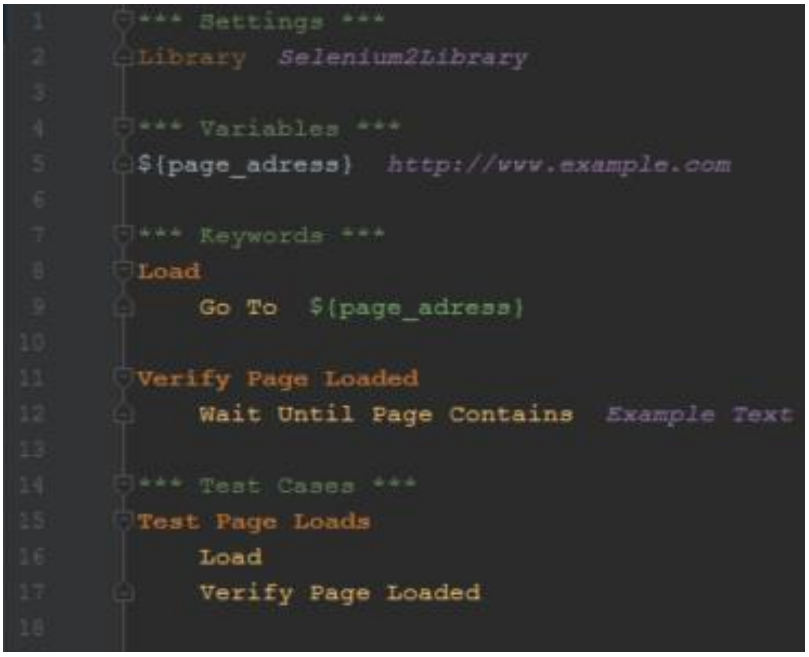
Testitapaukset (engl. test cases) kuvaavat jonkin yksittäisen asian testaamista (ks. kuvio 7). Esimerkkinä testitapauksesta on verkkokauppaan sisäänkirjautuminen ja varmistaminen, että toimenpide onnistuu. Testitapaukset on syytä pitää selkeästi luettavina ja pilkkoa ominaisuuksia kirjastoihin ja avainsanoihin. Lisäksi muuttujien käyttö on suositeltavaa, jolloin testien tulokset ovat helpompia lukea. Muuttujien käytöstä on hyötyä myös testien uudelleen käytettävyyden näkökulmasta. Testejä on mahdollista kirjoittaa useissa eri formaateissa kuten HTML, TSV, reStructuredText ja pelkkä teksti. Näistä pelkkä teksti on usein selkein vaihtoehto. (Robot Framework User Guide Version 3.0.2.)

```
33  Should be able to buy a regular item
34      [Tags]  Smoke
35      Login with valid credentials  ${LOGIN_NAME}  ${LOGIN_PASSWORD}
36      Open Shopping Cart Page
37      Clear Shopping Cart
38      Search for Single Product  ${TEST_PRODUCT1}
39      Open Shopping Cart Page Dropdown
40      Verify Item is in Cart  ${ITEM_NUMBER}
41      Proceed to Checkout page
42      Checkout Page Fill Info and Proceed
```

Kuvio 7. Esimerkki testitapauksesta

Testikokoelma

Testikokoelma (engl. test suite) muodostuu yksittäisestä testitiedostosta. Siihen kerätään yhteen samaa ominaisuutta testaavat testitapaukset. Siinä voidaan määritellä kaikille testitapauksille yhteiset alku- ja lopputilanteet. Lisäksi siinä voidaan määritellä asetuksia kuten kirjastot, muuttujat ja lähdetiedostot



Kuvio 8. Esimerkki testikokoelmasta

Yllä olevasta kuvasta voidaan havaita, että testikokoelma on mahdollista jakaa neljään eri loogiseen osaan (ks. kuvio 8). Settings (asetukset) osassa määritellään käytettävät kirjastot avainsanalla Library. Lisäksi siinä voidaan määritellä muita lähdetiedostoja käytettäväksi sekä ehdot aloitus ja lopetustilanteissa. Variables (muuttujat) osiossa voidaan määritellä muuttujia, joita voidaan uudelleen käyttää muissa osioissa. Muuttujan määrittely alkaa aina $-merkillä ja lista muuttujan @-merkillä. Keywords (avainsanat) osiossa puolestaan määritellään yksittäisiä toimenpiteitä kuten mene sivulle tai varmista että sivulla on jokin tietty teksti jne. Test Cases (testitapaukset) taas

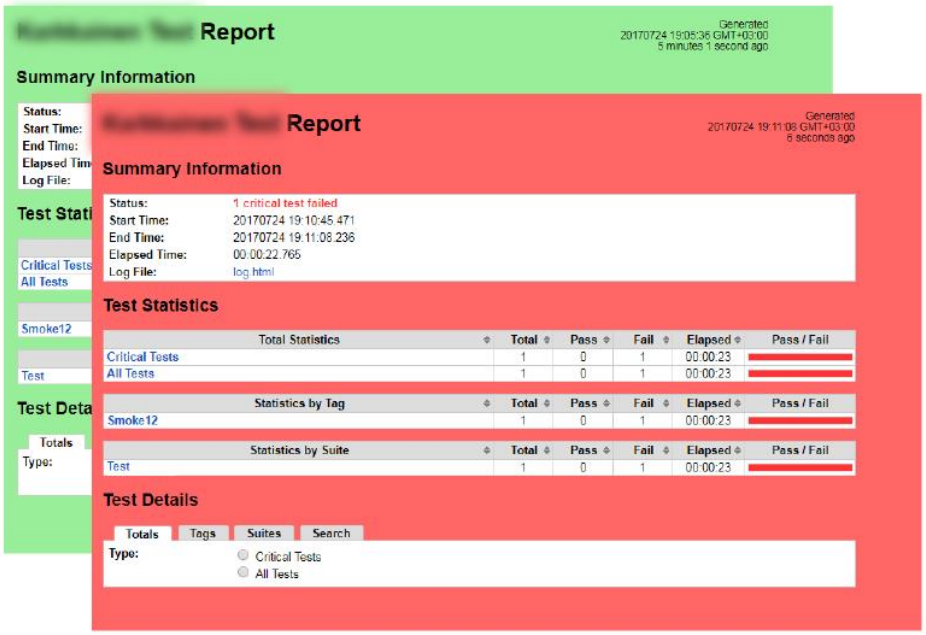
puolestaan rakentuvat näistä avainsanojen toiminnollisuuksista. (Robot Framework User Guide Version 3.0.2.)

5.4 Testien suorittaminen

Robot Frameworkin testien ajo suoritetaan komentoriviltä ja testien tulokset muodostuvat XML ja HTML muodoissa, joista jälkimmäistä voi tarkistella selaimesta käsin. Testien suorittamiseen käytetään komentoa: robot [options] data_sources. Vanhemmissa versioissa käsky on muuten sama, mutta se alkaa pythonin tapauksessa komennolla: pybot (Robot Framework User Guide Version 3.0.2.)

5.5 Testien ulostulo

Testien ajamisen jälkeen Robot Framework muodostaa 3 tiedostoa ajetuista testeistä: log.html, report.html ja output.xml. Log.html tiedosto on näistä kolmesta kaikista tärkein ja käytetyin, tätä tiedostoa tarvitaan joka kerta, kun tietoja halutaan tarkastella tarkemmin testin tuloksista. Report.html tiedosto sisältää paremman kokonaiskuvan ajetuista testeistä html muotoilulla. Report.html tiedosto on myös värikoodattu, jos kaikki testit onnistuvat sivu on vihreä. Jos taas jokin on mennyt pieleen, niin sivu on punainen (ks. kuvio 9). Report.html tiedosto sisältää myös linkit log.html tiedostoon, mikäli testejä pitää tarkastella tarkemmin. Output.xml tiedostossa on testi tulokset XML muodossa. (Robot Framework User Guide Version 3.0.2.)



Kuvio 9. Esimerkki testien tuloksista

5.6 Testikirjastot ja muut hyödylliset työkalut

Robot Frameworkin mukana tulee useita standardikirjastoja, joiden avulla voi mm. ottaa kuvakaappauksia, käsitellä päivämääriä, muokata merkkijonoja. Kirjastoja käytetään Settings-osiossa avainsanalla Library kirjastonnimi. Kuitenkin on olemassa useita ulkoisia kirjastoja, joita tarvitaan mm. Web-testaukseen. Näistä eräs suositumpimmista on nimeltään Selenium2Library. (Robot Framework User Guide Version 3.0.2.)

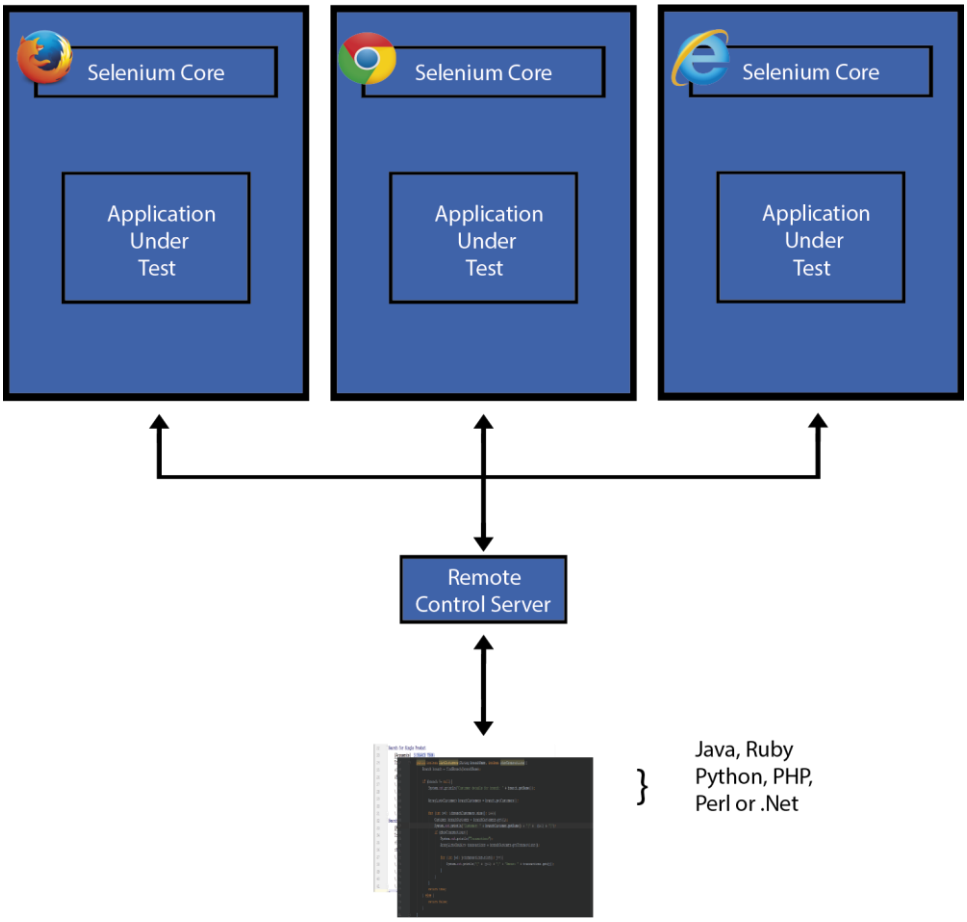
Selenium

Selenium on joukko erilaisia ohjelmointi työkaluja jotka helpottavat testi automaatiota. Sitä käytetään lähinnä testaamaan www-sovelluksia ja siihen voidaan sisältää erilaisia kirjastoja, kuten Robot Framework ja JUnit. Seleniumia voidaan käyttää useiden alustojen kautta (Windows, OSX, Linux) ja se tukee suurimpia selaimia kuten

Google Chrome, Internet Explorer, Mozilla Firefox, Safari ja Opera. Testejä voidaan ajaa tietokoneelta tai vaikka puhelimesta. Itse testit voidaan kirjoittaa usealla ohjelmointi kielellä kuten Java, Python, C#. Seleniumin mukana tulee useita työkaluja joista käyttäjä voi itse valita projektiin parhaiten sopivat työkalut. Erilaiset työkalut antavat uusia lähestymiskulmia ongelmien ratkontaan testaamisen automatisoinnissa. (Selenium Documentation, n.d, luku Introduction.)

Selenium RC

Selenium Remote Control on ollut mukana Seleniumin ensimmäisestä versiosta asti, kunnes Selenium WebDriver julkaistiin Selenium 2.0 kanssa. Siinä on toiminnallisuuksia jota Selenium 2.0 ei tue, kuten kyky ymmärtää useita ohjelmointikieliä (Java, Javascript, Ruby, PHP, C#) ja tuki lähes jokaiselle selaimelle. Selenium RC koostuu Selenium Serveristä ja asiakas kirjastoista. Serveri on osa testi ohjelman ja testattavan applikaation välillä, se saa komennot testi ohjelmalta ja suorittaa ne testattavaa applikaatiota vasten ja palauttaa tulokset käyttäjälle. Kirjastot tarjoavat käyttöliittymän ohjelmointikielien ja Serverin välille (ks. kuvio 10). (Selenium Documentation, n.d, luku Selenium 1 (Selenium RC).)



Kuvio 10. Selenium Remote Control arkkitehtuuri

Selenium WebDriver

Selenium 2.0: n isoin uusin lisäys on Selenium WebDriver. WebDriver on kehitetty korjaamaan ja parantamaan Selenium-RC API:n toiminnallisuuksia. Webdriver tarjoaa yksinkertaisemman ohjelmointi rajapinnan ja paremman tuen dynaamiselle nettisivuille.

WebDriver ei ole riippuvainen Selenium Serveristä toisin kuin Selenium-RC. WebDriver tekee suoria pyyntöjä selaimeen käyttäen jokaisen selaimen natiivia tukea automatisointiin. Tämä on suurin ero WebDriverin ja Selenium-RC:n välillä. Selenium-RC injektoi Javascript funktioita selaimeen, kun selain ladataan ja nämä funktiot ajetaan selaimen sisältä. WebDriver ei käytä tätä tekniikkaa, tämä tekee testien kirjoittamisesta ja ajamisesta nopeampaa.

Google Chromen kehitystyökalut

Kehitystyökalut (engl. developer tools) on suosittuun Google Chromeen selaimeen sisäänrakennettu ominaisuus, jonka avulla kehittäjät voivat tarkastella useita eri asiakaspuolen asioita. Kehitystyökalut mahdollistavat mm. kuvien, tyylien, skriptien tarkasteluun ja muokkaamisen välineitä. Kehitystyökalut saa Chromessa käyttöön navigointipalkista kohdasta More Tools ja valitsemalla Developer Tools kohdan tai yksinkertaisesti pikanäppäimellä ctrl + shift + i. (Barron 2015.)

Testaamisen kannalta yksi oleellisimmista työkaluista kehitystyökaluikkunassa on kohta elementit (engl. elements). Elementit kohdasta testaaja voi tarkastella www-sivun HTML rakennetta. Viemällä hiiren elementin päälle ja klikkaamalla sitä oikealla hiirenpainikkeella testaaja saa elementistä lisätietoja, kuten css selektori (engl. selector) tai Xpathin, joita käytetään testauksessa esimerkiksi määrittämään mitä elementtiä tulee klikata. Kehitystyökaluissa on myös kohta tyylit (engl. styles), jonka avulla testaaja voi tarkastella eri elementtien laatikkomalleja ja tyylejä ja näin esimerkiksi päätellä, miksi jokin elementti ei ole näkyvillä. (Barron. 2015.)

Kohdasta Network näkee, kuinka nopeasti eri tiedostot ovat ladanneet, minkä tyyppisiä ne ovat, minkä statuskoodin ne ovat antaneet ja missä järjestyksessä ne ovat ladanneet. Näitä tietoja voidaan käyttää selvittäessä pullonkauloja www-sivun latausvaiheessa. Aikajana (engl. Timeline) kohta puolestaan näyttää näiden resurssien taukset aikajanalla. Näin kehittäjät näkevät, missä resurssissa kestää kauiten ladata. Konsoli osio on myös kätevä testauksessa, sillä siitä näkee virhetilanteissa viestejä, mikäli www-sivun Javascriptissä, jokin ei toimi oikein. Kehitystyökalujen avulla on myös mahdollista tarkastella, miltä sivut näyttävät eri laitteilla ja miten sivujen responsiivisuus eli skaalautuminen eri laitteille toimii. (Barron. 2015.)

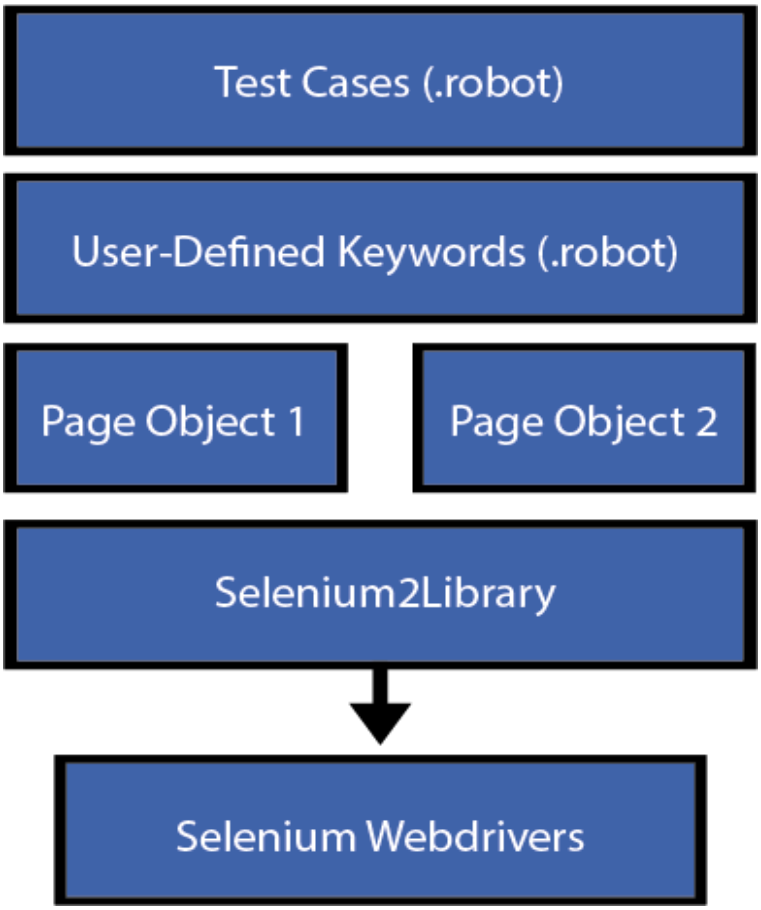
5.7 Robot Frameworkin käytänteet

Tiettyjä käytänteitä tulee asettaa testien kirjoittamiselle, jotta testien laatu pysyy hyvänä. Tärkeimmät asiat kirjoittaessa testejä Robot Frameworkilla ovat testien helppolukuisuus, ylläpidettävyys ja nopeus. Helppolukuisuus itse koodissa lisää myös tulosten selkeyttä ja helpottaa virheiden löytymistä. (Klärck. 2014.)

Muuttujien, testitapausten, testikokoelmien, kirjastojen ja avainsanojen tulee olla nimetty selkeästi ja kuvaavasti. Avainsanat alkavat tyypillisesti isolla kirjaimella ne on eroteltu toisistaan välilyönnillä. Pienempien osien kohdalla nimeämisen tulee olla niin hyvä, ettei kommentteja tarvitse käyttää, vaan nimen itsessään tulee kuvata toiminnallisuuttaan. Dokumentaatiota tulee puolestaan käyttää testikokoelmissa kuvamaan niiden taustaa ja tarkoitusta. Puolestaan testitapausten ja avainsanojen kohdalla dokumentointi on usein turhaa. (mt.)

Testitapauksia kirjoittaessa tulee pienimmät toimenpiteet pilkkoa avainsanoihin, jolloin rakenne pysyy helposti luettavana. Tällöin myös muut testitapaukset voivat käyttää samoja avainsanoja ja asioiden toistaminen vähenee. Testitapauksessa tulee myös olla tarkistuksia, eikä vain pelkästään asioiden tekemistä. Kuitenkin liiallinen tarkistusten kylväminen lähes joka komennon jälkeen on turhaa. Aloitus ja lopetusehtojen (engl. setup ja teardown) lisääminen testikokoelmiin lisää testitapausten suoritusnopeutta ja vähentää koodissa ilmenevää asioiden turhaa toistamista. Muuttujia tulisi käyttää aina, kun jokin sama arvo joudutaan kirjoittamaan kahteen eri paikkaan, jolloin muutoksen ilmetessä tarvitsee muuttaa vain yhtä osaa koodissa. Testidatassa ei saisi käyttää monimutkaista logiikkaa, vaan monimutkaiset toimenpiteet tulisi pilkkoa kirjastoihin ja kirjoittaa Pythonilla. Sleep komentoa tulisi vältellä viimeiseen asti, sillä se lisää koodin suoritukseen kuluvaa aikaa. Ennemmin tulisi käyttää Wait Until (engl. odota kunnes) -alkuisia komentoja. (mt.)

Eräs tehokas tapa kirjoittaa Robot Framework testejä on Page Objectien käyttö. Käytän tästä lähtien termistä lyhennettä PO. PO tunnetaan myös lisäksi termillä Page Resource, sillä Robot Framework ei ole oliopohjainen testiautomaatiokehys. PO perus idea on luoda jokaisesta sivusta olio ja näin kapseloida sivun testattavat ominaisuudet. (Using the Page Object Pattern with Robot Framework, n.d.)



Kuvio 11. Esimerkki Page Object-mallista

Yllä olevan kuvion mukaisesti ylimmällä tasolla on testitapauksia, jotka käyttävät käyttäjän määrittelemiä avainsanoja (ks. kuvio 11). Avainsanat puolestaan käyttävät PO:ta, jotka kuvaavat yksittäisiä sivuja tai sivun osia. Kuvitellaan tilanne, että meillä on 200 eri testitapausta, jotka testaavat toiminnollisuuksia laajasta verkkokaupasta. Verkkokaupan ostoskori nappulaa muutetaan, jolloin jokaisen testitapauksen avainsanoihin joudutaan tekemään muutos, jotka käyttävät tätä nappulaa. Tilanne on kuitenkin toinen PO:ta käytettäessä, jolloin muutos tehdään vain yhteen paikkaan. PO käyttö vähentää asioiden kirjoittamista useaan paikkaan ja selkeyttää laajojen testien rakennetta. (Using the Page Object Pattern with Robot Framework, n.d.)

6 Vanhojen testien korjaus

Tässä osassa määritellään asiakkaan x ylläpitotestien alkutilanne ja tavoitteet. Lisäksi osiossa perehdytään vanhoihin testeihin ja niiden parannukseen. Lopuksi korjataan vanhat testit ja muutetaan niiden rakenne selkeämmäksi ja ylläpidettävämmiksi.

Alkutilanne

Solteqin asiakkaan x verkkokauppaa ollaan päivittämässä uuteen versioon. Verkkokaupan versionpäivityksessä ei ole tiedossa mitään sellaisia asioita, joiden voisi sanota varmasti hajoavan päivittäessä. Tämän vuoksi verkkokaupan alustaa ei tulla tässä työssä käsittelemään. Vanhaan versioon on olemassa Robot Frameworkilla toteutetut automaattiset ylläpitotestit, mutta niitä ei ole hetkeen päivitetty, joten ne eivät toimi oikein. Lisäksi testit eivät ole riittävän kattavat, jolloin testitapauksia tulisi kirjata muutama lisää. Testit ovat tyypiltään ylläpitotestien lisäksi savutestejä.

Tavoitteena on kehittää yön aikana suoritettavat automaattiset ylläpitotestit. Testien tulee olla mahdollisimman ylläpidettävät, rakenteeltaan selkeät sekä tuloslokien tulee olla helposti tulkittavat. Lisäksi testeissä on tarkoitus hyödyntää PO-käytänteiden mukaista lähestymistapaa, missä jokaisesta sivusta tai sivun osasta muodostetaan oma olionsa, joka osaa tehdä tiettyjä toimenpiteitä.

Nykyiset testit testaavat verkkokaupan kustomoituja toimintoja, joita Solteq on kehittänyt verkkokauppa alustan päälle. Mitä enemmän verkkokaupan alustaa muokataan, sitä kattavampia testejä tulee tehdä. Ylläpito testauksella varmistetaan, ettei mikään vanha toiminnallisuus hajoa ja että uudet ominaisuudet toimivat oikein. Ylläpito testeillä pyritään parantamaan tuotteen laatua ja vähentämään kehitykseen kuluvaa aikaa, mikä puolestaan vähentää tuotteen kehitykseen kuuluvia kustannuksia.

6.1 Testien analysointi

Vanha testikokoelma on pilkottu kolmeen eri tiedostoon: testikokoelmaan, resursseihin ja itse tehtyyn funktiokirjastoon. Testikokoelma sisältää seitsemän eri testitapausta jotka ovat: kirjautuminen, normaalin tuotteen ostaminen, mittatuotteen ostaminen, tuotteen ostaminen määräalennuksella, kategorioiden varmentaminen, vara-

osien hakeminen, sekä varaosien ostaminen mallin mukaisesti. Testikokoelman testitapauksissa kutsutaan resurssitiedoston avainsanoja, jotka suorittavat varsinaisia testitoimintoja. Lisäksi monimutkaiset toimenpiteet on kirjoitettu funktiokirjastossa käyttäen Pythonia ja Selenium kirjastoa.

Vanhan testikokoelman testit on tehty osittain hyviä käytänteitä noudattaen, mutta testikokoelmassa on kuitenkin monia asioita, jotka voi tehdä paremmin ja selkeämmin. Vanha testikokoelma oli jaettu kolmeen eri tiedostoon, testien suorittamisen jälkeen Robot Framework tuottaa kolme loki tiedostoa ja lisäksi näihin liittyviä kuvia, joita otetaan testien kaatuessa ja testien lopuksi. Kaikkien testi tiedostojen sijaitseminen samassa paikassa tekee kansio rakenteesta sekavan. Järkevämpää on tehdä testikokoelmille, resursseille ja tuloksille kaikille omat kansionsa, jolloin tiedostot ja testien eri toiminnot olisi selkeämmin jaoteltu.

Testien suorittaminen on myös varsin hidasta. Tähän vaikuttaa mm. testeissä olevat SLEEP komennot, jotka pakottavat testejä odottamaan tietyn ajan. Robot Frameworkin hyvien käytänteiden mukaan SLEEP komennon käyttöä on pyrittävä välttämään, sillä se pakottaa testejä odottamaan, joka hidastaa testien suoritusnopeutta. Sen sijaan tulisi käyttää WAIT UNTIL alkuisia komentoja, jotka odottavat, kunnes jokin asia on ladannut tai valmis. Tässä projektissa kyseiset komennot eivät varsinaisesti haittaa, sillä verkkokaupan testejä suoritetaan pääosin öisin, jolloin aikaa on riittävästi. Kuitenkin kehitysvaiheessa kuluu turhaan aikaa, mikäli testien suorittaminen on liian hidasta. (Klärck. 2014.)

Vanhoissa testitapauksissa ei käytetty juurikaan muuttujia, mutta Robot Frameworkin hyviin käytänteisiin kuuluu kattava sekä hyvin selkeä muuttujien ja avainsanojen käyttö. Tämä tekee testin loki tiedostosta paljon helppolukuisemman ja samoja asioita ei toisteta turhaan. Vanhoissa testeissä käytettiin myös paljon kommentteja kuvaamaan eri testitapausten toiminnallisuuksia. Kun muuttujia ja avainsanoja on tarpeeksi ja ne ovat hyvin nimettyjä, niin kommentteja ei enää tarvita kuvaamaan testien toimintoja. Vanhat testitapaukset eivät käyttäneet myöskään TEST SETUP komentoa, tämän vuoksi samat asiat toistettiin jokaisen testitapauksen alussa turhaan. TEST SETUP komento alustaa testitapaukset alkamaan tietyillä ehdoilla, esim. Avaa selain, mene sivulle amazon.com (ks. kuvio 12). (Klärck. 2014.)

```
*** Variables ***
${WEBSITE}  https://www.amazon.com/

*** Keywords ***
Begin Web Test
    [Arguments]  ${WEBSITE}
    Open Browser  about:blank  chrome
    maximize browser window
    Go To  ${WEBSITE}



Should be able to verify bought products
    [Setup]  Begin Web Test
    #MORE CODE...
```

Kuvio 12. TEST SETUP komennon käyttö

Vanhoissa testitapauksissa käytettiin avainsana pohjaista syntaksia. Kaikki sivut ja niiden eri toiminnot olivat kuitenkin pakattuna kahteen eri tiedostoon. Yksi Robot Frameworkin hyvistä käytänteistä on käyttää PO:ita. PO:iden avulla sivujen loogiset osat jaetaan omiin olioihin, esim. Navigaatio palkki. Tämä auttaa selkeyttämään virheen löytämistä testilokeja tutkiessa ja lisäksi kehitysvaiheessa on helpompi hahmottaa testien eri toiminallisuudet. (Using the Page Object Pattern with Robot Framework, n.d.)

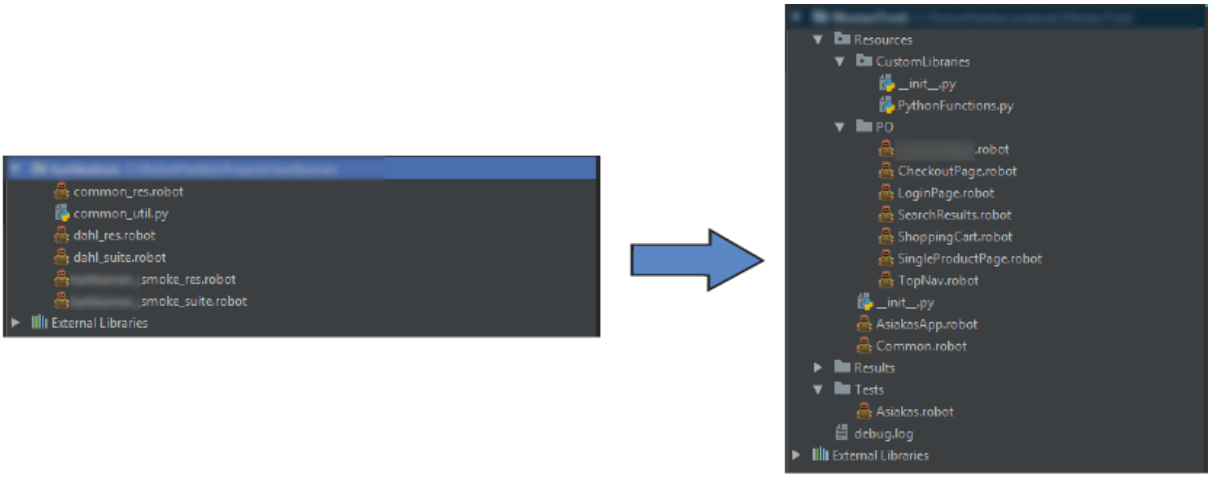
Vanhojen testien itsekirjoitettu funktio kirjasto toimi hyvin, sitä ei tarvinnut muokata. Funktio kirjastoa käytetään toteuttamaan monimutkaisia testitoimenpiteitä. Yksi esimerkki on yksittäiseen ostettavan tuotteen arpominen verkkokaupan hakutuloksista.

Testien suoritukseen kului aikaa noin kahdeksan minuuttia ja näistä testeistä noin puolet ajettiin onnistuneesti läpi. Itse testiloki on varsin sekava, etenkin sellaiselle henkilölle joka ei ole testejä kirjoittanut ja näin ollen luettavuus sekä ylläpidettävyys testeissä on huono. Näin ollen testien rakennetta, luettavuutta, suoritusnopeutta ja ylläpidettävyyttä tulee parantaa. Lisäksi tulee selvittää minkä takia nykyiset testit eivät mene onnistuneesti läpi.

6.2 Testien korjaus

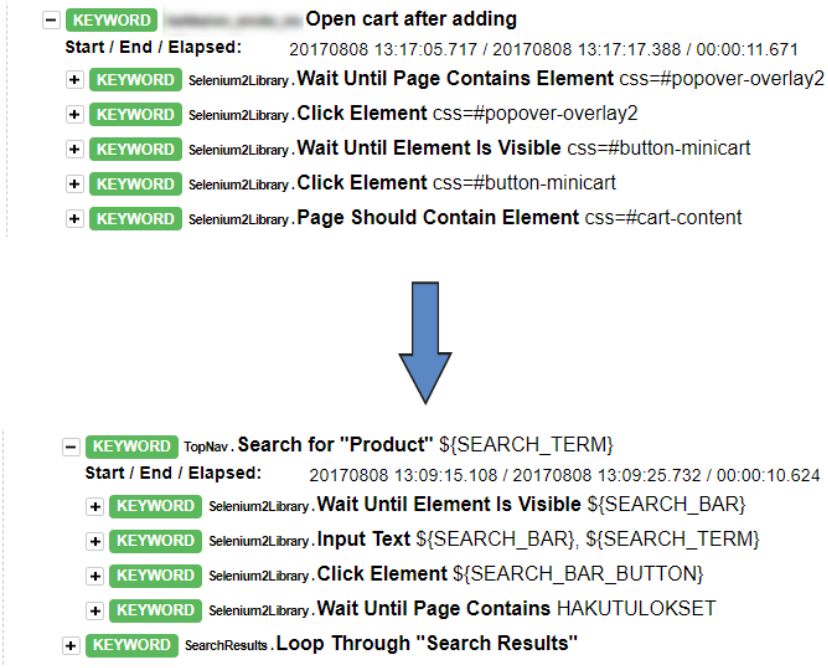
Vanhojen testien korjaaminen aloitettiin tekemällä täysin uusi projekti. Vanhasta testikokoelmasta otettiin mukaan testitapaukset ja muutamia siihen liittyviä muuttujia,

sekä itsetehty funktio kirjasto. Aiemmissa testeissä ei ollut kansio rakennetta, joten testeille rakennettiin kolme kansiota jakamaan testien toimintoja loogisiin osiin. Nämä kansiot ovat nimeltään: Tests, Results ja Resources. Tests kansiossa on testikokoelma, joka sisältää testitapaukset. Results kansioon tallentuu kaikki Robot Frameworkin tuottamat testiloki tiedostot ja niihin liittyvät kuvat. Resources kansiossa on funktio kirjasto ja alakansio nimeltä PO. PO kansio nimensä mukaisesti sisältää verkkosivun osia eli Page Objecteja (ks. kuvio 13).



Kuvio 13. Vanha kansiorakenne vs. uusi kansiorakenne

Page Objecteiksi muuttaminen aloitettiin jakamalla testien toiminnallisuuksia loogisiin osiin. Esim. TopNav PO, SearchResults PO, LoginPage PO. PO:n ei tarvitse olla välttämättä kokonainen sivu vaan siihen soveltuu hyvin mm. ylhäällä oleva navigointi palkki tai vaikka verkkokaupan ostoskori. Jakamalla testit näihin loogisiin osiin lokitiedostoista tulee selkeät ja virheen etsiminen helpottuu huomattavasti (ks. kuvio 14). (Using the Page Object Pattern with Robot Framework, n.d.)



Kuvio 14. Vanha loki vs. uusi loki

Vanhat kovakoodatut tiedot muutettiin siirtämälle ne selkeästi nimettyihin muuttujiin ja niiden nimet muutettiin paremmin kuvaaviksi. Tämä parantaa lokin luettavuutta entisestään sekä vähentää koodissa esiintyvää toisteliaisuutta (ks. kuvio 14). Käyttämällä muuttujia samojen asioiden uudelleen kirjoittamiselta vältytään. Vanhoissa teisteissä meni reilusti aikaa useaan SLEEP-komentoon, jotka korvattiin WAIT UNTIL–alkuisilla komennoilla. Aikaisempi komento odotti aina tietyn määritellyn ajan sekunteina ennen etenemistä. Puolestaan uusi komento etenee heti, kun asia tai toiminto on saatu ladattua valmiiksi. Vanhan testikokoelman jokaisessa testi tapauksessa käytettiin komentoa SET SELENIUM SPEED 1s, tämä lisää jokaiseen kutsuttavaan komentoon yhden sekunnin odotusajan. Komento on kätevä, mikäli halutaan tarkkailla testien kulkua näytöllä, mutta tuotanto käytössä se hidastaa merkittävästi testien suorittamista. Test Setup ja Test Teardown määrittelyt testikokoelmaan nopeuttavat testien suorittamista. Kuitenkaan vanhoissa testeissä ei käytetty Test Setup komentoa, vaan jokainen testi tapauksessa on toistettu samoja komentoja alussa turhaan. (Klärck. 2014.)

Testien rakenteen korjaamisen jälkeen selvitettiin, minkä takia vanhat testit eivät mene läpi. Yleisenä ongelmana useassa testissä oli ostoskorin avaaminen tuotteen lisäämisen jälkeen. Verkkokauppa alustaan oli lisätty oma kustomoitu mini ostoskori toiminnallisuus. Tämä ostoskori avautuu joka kerta kun tuote lisätään. Mini ostoskori aukesi varsinaisen ostoskorin päälle, joka puolestaan esti ostoskoriin etenemisen. Tämän vuoksi testit siirrettiin käyttämään mini ostoskoria suoraan ostoskori sivulle etenemiseksi lähes kaikissa testitapauksissa. Normaaliin ostoskorin etenemiselle ilman mini ostoskoria kirjoitettiin oma testitapaus.

Toinen yleinen ongelma testeissä oli, jos tuotetta ei ollut saatavilla verkkokaupasta. Tämä itsessään riitti kaatamaan testi tapauksen sen loppu vaiheilla. Ongelma korjattiin tekemällä ns. älykäs tuotteen valitseminen, joka tarkastaa ensin onko tuotetta saatavilla. Mikäli tuotetta ei ole saatavilla, palataan hakutuloksiin ja arvotaan uusi tuote, kunnes löytyy tuote, jota on saatavilla.

Yksi testitapaus liittyi varaosien etsimiseen. Siinä varaosaa haettiin sattumanvaraisesti valmistajan ja tuotteen merkin mukaan. Ongelmia tuotti se, ettei läheskään kaikilla merkeillä ollut tuotteita saatavilla. Tämä johti testitapauksien kaatumiseen tyhjien hakutuloksien takia. Ongelma korjattiin hakemalla tuotetta, jota on varmasti saatavilla projekti päällikön ohjeistusten mukaan.

Muutaman testitapauksen ongelman ratkomiseen rakennettiin itse tehtyjä python funktioita, esimerkiksi varaosien arpominen. Lopuksi testit saatiin toimimaan oikein, rakenne on siistimpi ja lokit ovat helppolukuisempia. Testien suoritusaika väheni kahdeksasta minuutista kahteen minuuttiin, mikä on selvä parannus nopeudessa. Seuraavaksi vuorossa oli uusien testitapauksien suunnitteleminen ja toteutus, jotta ylläpitotesteistä saadaan tarpeeksi kattavat.

7 Uusien testien toteutus ja automaatioputken asetus

Tässä kappaleessa käydään läpi uusien testien suunnittelu ja toteutus prosessi. Lopuksi testit asetetaan Jenkins automaatioputkeen.

7.1 Uusien testitapausten suunnittelu ja toteutus

Verkkokaupan alustaan oli tehty miniostoskori ja tämän toimintoja piti testata. Miniostoskori on oma looginen osansa, joten aivan aluksi Miniostoskorista tehtiin uusi PO luokka. Kun PO luokka oli valmis selvitettiin kehitystiimin kanssa, mitä asioita mini ostoskorin toiminnoista halutaan testata. Asiat jotka tulivat esille: Tuotteiden lisääminen mini ostoskoriin, ostoskorin tyhjentäminen tuotteista, ostoskorin sulkeminen, mini ostoskorista varsinaiselle ostosivulle eteäminen ja sen varmentaminen.

Mini ostoskoriin kuuluu muutamia staattisia elementtejä, jotka ovat usein käytössä, kuten tuotteen poistamiseen ostoskorissa käytettävä raksi elementti. Nämä elementit nimettiin selkeästi jotta koodi on selkeämpi lukea ja toisteliaisuus vähenee (ks. kuvio 15). (Klärck. 2014.)

```
*** Variables ***
${MINI_REMOVE_ITEM}   css=#mini-shoppingcart-products > div.products > div > a.removelink > i
${MINI_KASSALLE_BUTTON}    css=#mini-shoppingcart-overlay > div:nth-child(1) > div.go-on-button > a
${MINI_CART_CLOSE_BUTTON}   css=#mini-shoppingcart-overlay > div:nth-child(1) > div.close-button > a > i
```

Kuvio 15. Miniostoskorin muuttujia

Avainsanojen ja muuttujien suunnitelussa testitapauksessa tuli ottaa huomioon nimien samankaltaisuus normaalin ostoskorin testitapausten kanssa. Jokaiseen miniostoskorissa käytettävään avainsanaan lisättiin viittaus "Mini Shopping Cart". Tällä vältytään ristiriidoilta testi lokeja tutkiessa.

Ensimmäinen asia jota lähettiin toteuttamaan oli tuotteen lisääminen ostoskoriin. Tuotteen lisäämällä ostoskoriin se lisäytyy myös automaattisesti mini ostoskoriin, tähän toimintoon löytyi jo valmiina avainsana SingleProductPage PO sivulta Add Product to Cart. Avainsanaan lisättiin toiminto, joka ottaa automaattisesti tuotenumeron tuotteesta talteen globaaliin muuttujaan ${ITEM_NUMBER}. Tätä tuotenumeroa

voidaan hyödyntää ostosivulle edetessä varmentamaan, että kyseinen tuote löytyy myös ostosivulta kun sinne edetään.

Seuraavana oli vuorossa mini ostoskorin sulkeminen, tämä on hyvin yksinkertainen toiminto joille tehtiin oma avainsana Close Mini Shopping Cart Window (ks. kuvio 16).

```
Close Mini Shopping Cart Window
    Wait Until Element is Visible  ${MINI_CART_CLOSE_BUTTON}
    Click Element  ${MINI_CART_CLOSE_BUTTON}
```

Kuvio 16. Miniostoskorin sulkeminen

Yksi testattavista toiminnoista oli ostosivulle eteneminen miniostoskorin kautta. Kyseessä on käytännössä yhden nappulan klikkaaminen ja varmennus, siitä että siirtyminen sivulle onnistui. Tähän tehtiin avainsana: Proceed to "Checkout Page" Mini Shopping Cart. Verifikaatioon siirtymisestä testi tapauksessa käytetään ylempänä mainittua ${ITEM_NUMBER} muuttujaa. ShoppingCart PO johon liikutaan mini ostoskorista on ns. Ostosivu, se sisältää avainsanan Verify "Item" in Cart. Tämä tarkistaa että ostosivulta löytyy tuotenumerolla esiintyvä tuote.

Mini ostoskorin testaaminen onnistui hyvin pelkän Robot Frameworkin toiminnoilla, kunnes piti tyhjentää mini ostoskori useammasta kuin yhdestä tuotteesta. Tämä oli monimutkaisempi toiminto ja sitä varten piti kirjoittaa oma funktio funktiokirjastoon ja ottaa se käyttöön ostoskorin tyhjennystä varten. (Klärck. 2014.)

Funktion Delete All from Mini Shopping Cart tarkoitus on tutkia lukeeko ostoskorissa "Ostoskori on tyhjä", mikäli tämä ei toteudu niin klikataan tuotteiden poisto linkkiä kunnes ostoskori on tyhjä (ks. kuvio 17).

```
def delete_all_from_mini_shopping_cart(empty_text):
    driver = get_driver()
    if not (empty_text.encode('utf-8') in driver.page_source.encode('utf-8')):
        buttonlist = driver.find_elements_by_css_selector('a.removelink  > i')
        while len(buttonlist) > 0:
            buttonlist[0].click()
            time.sleep(5)
            buttonlist = driver.find_elements_by_css_selector(
                'a.removelink > i')
```

Kuvio 17. Python-funktio esineiden poistamiseksi miniostoskorista

Kun kaikki testiin liittyvät toiminnot oli saatu valmiiksi rakennettiin itse testitapaus valmiiksi. Testissä liikutaan ensiksi lisäämään kaksi erilaista tuotetta ostoskoriin, tämän jälkeen tuotteet poistetaan mini ostoskorista. Kun tuotteet on saatu poistettua ja varmistettu että ostoskori on tyhjä, lisätään ostoskoriin vielä yksi tuote ja liikutaan maksusivulle mini ostoskorin kautta. Maksusivulla tapahtuu vielä viimeinen tarkistus siitä, että tuote on onnistuneesti siirtynyt miniostoskorista maksusivulle (ks. kuvio 18).

```
Should be able to shop with Mini Shopping Cart
    [Tags]  Shop
    [Setup]  Begin Web Test Blank
    Go To  ${TEST_PRODUCT_DIRECT_LINK}
    Add Product to Cart
    Go to  ${TEST_PRODUCT_DIRECT_LINK2}
    Add Product to Cart
    Delete all from mini shopping cart  Ostoskorisi on tyhjä
    Verify Mini Shopping Cart is Empty
    Close Mini Shopping Cart Window
    Add Product to Cart
    Proceed to "Checkout page" Mini Shopping Cart
    ShoppingCart.Verify "Item" in cart  ${ITEM_NUMBER}
```

Kuvio 18. Miniostoskori testitapaus

7.2 Automaatioputken asetus

Solteqilla on käytössään testipalvelimia, joihin on asennettu Jenkins. Kokonaisuus mahdollistaa testien rinnakkaisajon eli useaa Jenkinsin jobia voidaan ajaa yhtäaikaisesti. Lisäksi se mahdollistaa Master Slave arkkitehtuurin mukaisen testien yhtäaikaisen suorittamisen eri palvelimilla. Ympäristö sisältää myös useita työnkannalta oleellisia liitännäisiä kuten Hipchat, Robot Framework ja Mercurial liitännäiset.

Kun testitapaukset oli saatu kirjoitettua valmiiksi olivat ne valmiita Jenkinsiin siirtoon. Ennen Jenkinsiin siirtoa siirrettiin testiskriptit versionhallintaan, jotta Jenkinsillä olisi pääsy niihin. Jenkinsissä oli valmiina vanha jobi testeille, mutta uuden testausympäristön myötä sen aiemmat asetukset eivä olleet enää toimivia testien suorittamiseksi. Tämän vuoksi Jenkinsiin asetuksiin ja testiskripteihin täytyi tehdä muutoksia.

Käytännössä testit suoritetaan kehitysympäristössä entiseen tapaan, mutta Jenkinsissä pybot-komennon yhteydessä annetaan parametri HUB:TRUE, joka mahdollistaa ympäristön ulkopuolisen palvelimen käytön testejä suorittessa.

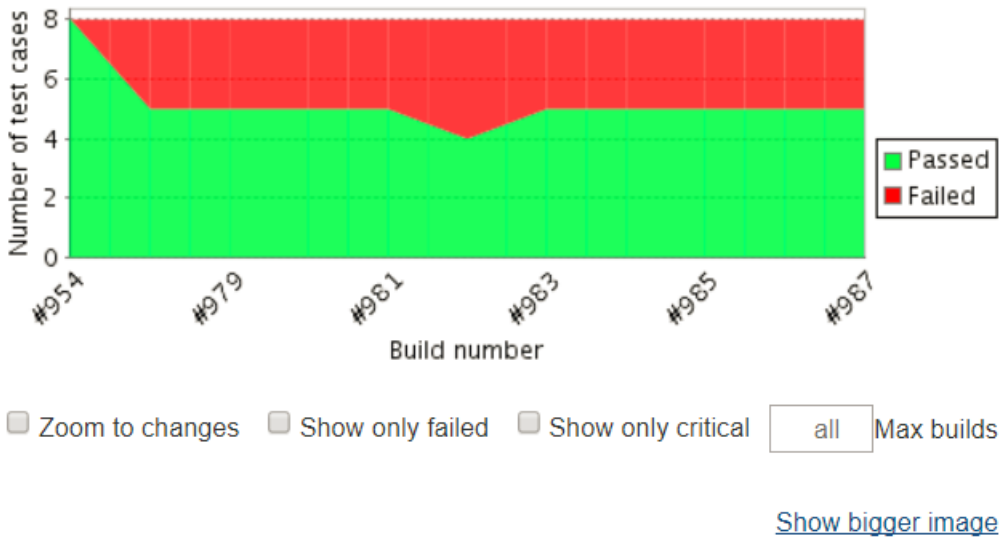
Testit asetettiin aluksi suoriutumaan, joka arkipäivä klo 04:00 hieman yöllisen testi kaupan deployn jälkeen. Testit voidaan laittaa juoksemaan myös triggerin avulla, tämä todettiin testien juoksemisen kannalta paremmaksi vaihtoehdoksi. Triggeri asetettiin ajamaan testit, kun testi kaupan deploy on suoritettu onnistuneesti. Käyttämällä triggeriä poistetaan testien riippuvuus kellonajasta. Tämä mahdollistaa testien automaattisen juoksun, jos deploy halutaan tehdä kesken työpäivää tai jos deployn ajaminen myöhästyy. Tämä lähestymistapa on lähempänä jatkuvan integraation ideologiaa. (Eficode Quick Guide: DevOps, 14)

Testejä yritetään suorittaa maksimissaan kolme kertaa, mikäli ne eivät ensimmäisellä tai toisella suorituskerralla mene kokonaan läpi. Tämä tehdään sen vuoksi, että testiympäristön kaupan välimuisti on heti deployn jälkeen aivan tyhjä ja se saattaa aiheuttaa testien kaatumista sivuston hitauden vuoksi. Testien suorittaminen täyttää kaupan välimuistia, mikä nopeuttaa verkkosivun toimintaa. Tämä on hyvä ominaisuus aamulla töihin tuleville kehittäjille tai mikäli deploy on ajettu kesken työpäivän, sillä testikauppa on testien suorittamisen seurauksena nopeutunut.

Testien suoritumisesta oli aiemmassa toteutuksessa lähetetty kaikille kehitystiimin jäsenille sähköpostia, jossa näkyy statistiikkaa yön aikana suoritetuista testeistä. Se päätettiin kuitenkin korvata Hipchat-lisäosalla, jolla voi laittaa testien suorituksesta viestiä automaattisesti kehitystiimin käyttämään Hipchat-ryhmään. Viestiä laitetaan jokaisesta epäonnistuneesta testien suorituksesta ja siinä näytetään kuinka moni testi epäonnistui suhteessa testien kokonaismäärään. Viestiä on myös mahdollista laittaa testien alkaessa, päättyessä, keskeyteytyessä jne., mutta näille ominaisuuksille ei koettu tarvetta, sillä se olisi turhaan täyttänyt tiimin chättiä.

Testien suorittaminen

Jenkins jobin etusivulta näkee kätevästi, kuinka hyvin testit ovat viime aikoina suoriutuneet (ks. kuvio 19). Tämä statistiikka on tärkeää, kun testejä lähdetään Jenkinsiin laiton jälkeen hiomaan siten, että ne suoriutuvat mahdollisimman hyvin pitkällä aikavälillä. Testejä suorittaessa huomattiin, että varaosakauppaan liittyvät testit eivät aina suoriudu oikein, johtuen kolmannen osapuolen API-hitaudesta testiympäristössä. Näiden testien suoritukseen tulee varata normaalia enemmän aikaa eri komentojen välillä. Myös nopeuserot testien kehitysympäristön sekä Jenkinsin käyttämän ympäristön välillä vaihtelevat, jolloin testit vaativat pientä hienosäätöä suoriutuakseen oikein.



Kuvio 19. Testien suoritumistrendi

Asiakkaan x tiimin projektipäällikön pyynnöstä toteutettiin liite testitapauksista, jossa avataan tarkemmin testien käyttötarkoitus (ks. liite 1). Liitteessä käydään tarkkaan läpi mihin tarkoitukseen, millä tavalla, miten ja miksi testi on kirjoitettu. Liite on tärkeä testien ymmärtämisen kannalta ja myös siksi, että testejä on helpompi ylläpitää, kun niistä on tehty yksiselitteiset kuvaukset. Myös testien laajentamiseen kannalta liite testitapauksista on tärkeä.

8 Tutkimustulokset ja johtopäätökset

Tässä osiossa pyritään vastaamaan tutkimuskysymyksiin. Jokaisen tutkimuskysymyksen alle on koostettu lyhyt tiivistelmä ja niiden tarkoituksena on selvittää millaisiin tutkimustuloksiin ja johtopäätöksiin on päädytty.

Miten automaattinen ylläpitotestaus toteutetaan asiakkaan X verkkokauppaan?

Ennen työn aloittamista asiakkaan X verkkokauppaan oli olemassa vanhat ylläpitotestit, jotka eivät toimineet. Ne oli toteutettu Robot Frameworkilla ja testien suorittamisesta ja automaatiosta vastasi Jenkins. Työssä onnistuttiin korjaamaan vanhat testitapaukset sekä niihin lisättiin yksi uusi testitapaus. Lisäksi testien selkeytä, ylläpidettävyyttä sekä suorituskykyä saatiin parannettua. Testit asetettiin osaksi Jenkins automaatioputkea, jolloin ne suoriutuvat, joka yö tiettyyn kellonaikaan. Jenkins mahdollistaa myös testien tuloksien helpon seuraamisen sekä ilmoitukset tilanteessa, jossa jokin testi ei mene läpi.

Miten Robot Frameworkilla kirjoitetaan selkeitä, ylläpidettäviä ja nopeita testejä?

Kirjoittaessa testejä Robot Frameworkilla on useita seikkoja, jotka tulee ottaa huomioon, jotta testeistä saadaan mahdollisimman selkeitä ja ylläpidettäviä.

Testiprojekti kannattaa kansioiden tasolla pilkkoa useampaan osaan, jolloin tietyn tyyppiset asiat ovat omissa kansioissaan esim. ulkoiset kirjastot ja lokien tulokset. Muuttujien, avainsanojen, testitapausten jne. nimeäminen tulee olla yksiselitteisiä ja helposti ymmärrettävässä muodossa. Testikokoelmat tulisi dokumentoida kattavasti, mutta testitapaukset ja niiden sisältämät avainsanat tulisi kirjoittaa niin selkeillä termeillä, että ne eivät vaadi dokumentaatiota.

Muuttujia tulisi käyttää mahdollisimman paljon, jolloin vähennetään asioiden tarpeetonta toistamista ns. ”kovakoodausta". Monimutkainen testilogiikka tulisi siirtää kirjastoihin aina, kun se on vain mahdollista. Testitapauksissa tulee olla välitarkastuksia, jolloin vian selvittäminen helpottuu. Lisäksi nämä tarkastukset kannattaa toteuttaa Wait Until-alkuisilla komennoilla, jolloin testien suoritusnopeus kasvaa.

Työssä päätettiin käyttää PO-käytänteiden mukaista tyyliä testejä kirjoittaessa. PO-käytänteistä on etenkin hyötyä, silloin kun testitapauksia on suuri määrä, koska koodin itsensä toistaminen vähenee merkittävästi. Työssä ei ole kovin montaa testitapausta, mutta mikäli tulevaisuudessa testitapauksia tehdään lisää, tarjoaa käytetty malli, sille hyvän pohjan koodin tasolla.

Kuinka testit saadaan integroitua osaksi Jenkinsin automaatioputkea?

Solteqilla oli valmiina testipalvelimet, joihin oli asennettu Jenkins ja työn kannalta oleelliset lisäosat. Tarkoituksena oli selvittää, miten testit asetetaan Jenkinsiin, sekä siihen millaisia toiminnollisuuksia se tarjoaa testien hallitsemiseksi. Testit saatiin siirrettyä onnistuneesti Jenkinsiin ja niitä ajetaan, joka yö tuotantoon viennin jälkeen. Mikäli jokin testikokoelman testeistä ei mene läpi siitä tulee viestiä kehitystiimille. Testien siirtäminen Jenkinsiin sujui suhteellisen helposti pienin koodinmuutoksin testien asetuksissa sekä sitä suorittavassa komennossa.

9 Pohdinta

Opinnäytetyössä saatiin korjattua vanhat testitapaukset, luotua uusia, ja paranneltua niiden rakennetta ja suorituskykyä. Työkaluna tähän sovellettiin Robot Framework testityökalua, joka on yrityksellä yleisesti käytössä. Lisäksi testit asetettiin Jenkins automaatioputkeen suoriutumaan automaattisesti jokaisen tuotantoympäristöön viennin jälkeen.

Tutkimuksen teko ja testien rakentaminen sujuivat ilman suurempia ongelmia. Robot Framework on helposti ymmärrettävä testi rajapinta ja ohjelmointia osaavalle henkilölle helppokäyttöinen. Linja-aho (2017, 29) on myös opinnäytetyössään todennut Robot Frameworkin olevan vaivaton testejä kirjoittaessa. Lisäksi monimutkaisempien

funktioiden kirjoittamiseen käytetty Python-ohjelmointikieli on helposti ymmärrettävää ja sillä on nopea kirjoittaa erilaisia toiminnallisuuksia. Tämän lisäksi Internetistä löytyi kattava dokumentaatio testitapauksista ja komennoista, joista oli suuri apu testejä kirjoittaessa. Kuitenkin Jenkinsin käyttöönotossa ilmeni ongelmia, sillä uusien testipalvelimien toimituksen myöhästyminen viivästytti opinnäytetyön valmistumista ja tämän vuoksi työ jouduttiin keskeyttämään hetkellisesti.

Tutkimuksen tuloksista on hyötyä toimeksiantajalle, sillä toimivien ylläpitotestien avulla saadaan havaittua virheet helpommin ja näin parannettua verkkokaupan laatua. Automatisoidut ylläpitotestit eivät kuitenkaan poista tarvetta perinteiseltä manuaaliselta testaukselta. Ne vaativat ylläpitoa ja resursseja siinä, missä perinteinen käsin testauskin. Otollisimpia käyttökohteita niille ovat useasti toistettavat testitapaukset sekä vanhojen ominaisuuksien toiminnan varmentaminen.

Saaduista tuloksista on hyötyä, mikäli käytetään samoja työkaluja testien toteutukseen. Tulokset eivät ole suoraan yleistettävissä, mikäli testaus suoritetaan eri välineitä käyttäen. Myös käytetty PO-malli, jossa sivusto pilkotaan loogisiksi olioiksi on vain yksi tapa järjestellä ohjelmakoodi. Tämä tapa ei ole perinteinen malli kirjoittaa testejä Robot Frameworkilla, mutta siitä on hyötyä etenkin, jos testitapauksia on suuri määrä. Työtä voidaan kuitenkin käyttää oppaana tilanteessa, jossa on tarve kehittää testitapauksia tai sen automaatiota työssä käytetyillä työvälineillä.

Tutkimus suoritettiin varsin pienellä määrällä testitapauksia. Tutkimista voisi jatkaa suuremmalla määrällä testitapauksia ja tarkastella sitä, miten laajempia testikokoelmia toteutetaan ja ylläpidetään tehokkaasti. Robot Frameworkin ja Jenkinsin lisäksi on olemassa useita muita eri testityökaluja ja sovelluskehyksiä. Lisätutkimusta voisi siis tehdä muista testien toteutusvälineistä ja esimerkiksi vertailla niiden käytettävyyttä ja suorituskykyä. Lisäksi opinnäytetyössä ei pureuduttu syvällisemmin Jenkinsin käyttöön ja konfigurointiin. Lisätutkimusta olisi siis mahdollista tehdä eri Jenkinsin osa-alueista mm. käyttöönotosta ja prosessien automatisoinnista.

Lähteet

Barron, B. 2015. Why You Should Start Using Chrome Developer Tools Right Now. Blogi postaus Elegantthemes www-sivulta.Viitattu 8.8.2017. https://www.elegantthemes.com/blog/resources/why-you-should-start-using-chrome-developer-tools-right-now.

Berg, A. 2012. Jenkins Continuous Integration Cookbook. Olton, Birmingham, GBR: Packt Publishing. Viitattu 25.6.2017. http://www.ebrary.com.

Black Box Testing. N.d. Dokumentti Software Testing Fundamentals www-sivulla. Viitattu 12.7.2017. http://softwaretestingfundamentals.com/black-box-testing/.

Cauldwell, P. 2008. Code Leader: Using People, Tools, and Processes to Build Successful Software. Birmingham: Wrox. Viitattu 15.7.2017. library.books24x7.com.

Certified Tester Foundation Level Cyllabus. 2011. Dokumentti ISTQB www-sivuilla. Viitattu 24.7.2017. http://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html4.

Eficode Quick Guide: DevOps. N.d. Efficoden julkaisema opas. Viitattu 14.2.2018. https://cdn2.hubspot.net/hubfs/2714969/Documents/Eficode-English-Devops-Guide.pdf.

Fowler, Martin. 2006. Continuous Integration. Artikkeli Fowlerin www-sivuilla. Viitattu 15.7.2017. http://martinfowler.com/articles/continuousIntegration.html.

Gray Box Testing. N.d. Dokumentti Software Testing Fundamentals www-sivulla. Viitattu 12.7.2017. http://softwaretestingfundamentals.com/gray-box-testing/.

GUI Testing: Complete Guide. N.d. Artikkeli guru99 www-sivulta. Viitattu 26.7.2017. https://www.guru99.com/gui-testing.html.

Hambling, B. 2015. Software Testing: An ISTQB-BCS Certified Tester Foundation Guide, Third Edition. BCS julkaisema opas. Viitattu 11.7.2017. library.books24x7.com.

Kananen, J. 2015. Kehittämistutkimuksen kirjoittamisen käytännön opas. Jyväskylän ammattikorkeakoulu.

Kasurinen, J. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

Klärck, P. 2014. Robot Framework Dos And Don'ts. Robot Frameworkin pääkehittäjän diaesitys www-sivulla. Viitattu 3.2.2018. https://www.slideshare.net/pekkaklarck/robot-framework-dos-and-donts.

Linja-aho, P. 2017. Testausautomaatio Robot Frameworkilla. Opinnäytetyö, AMK. Turun ammattikorkeakoulu, Tietotekniikan koulutusohjelma. Viitattu 14.4.2018. https://www.theseus.fi/bitstream/handle/10024/137804/Linja-aho_Petri.pdf?sequence=1&isAllowed=y.

LTS Release Line. N.d. Opas Jenkinsin www-sivuilla. Viitattu 18.7.2017. https://jenkins.io/download/lts/.

Myers, G., Sandler, C., & Badgett. 2012. The Art of Software Testing. 3rd.ed.

New Jersey: John Wiley & Sons, Inc. Viitattu 23.6.2017. library.books24x7.com.

Robot Framework User Guide Version 3.0.2. N.d. Opas Robot Frameworkin käyttöön työkalun www-sivulla. Viitattu 10.7.2017. http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

Saurabh. 2016a. What is Jenkins? | Jenkins For Continuous Integration | Edureka. Blogipostaus Edurekan www-sivuilla. Viitattu 14.7.2017. https://www.edureka.co/blog/what-is-jenkins/?utm_source=blog&utm_medium=left-menu&utm_campaign=install-jenkins.

Saurabh. 2016b. Jenkins Tutorial | Continuous Integration Using Jenkins | Edureka. Blogipostaus Edurekan www-sivuilla. Viitattu 14.7.2017. https://www.edureka.co/blog/jenkins-tutorial/?utm_source=blog&utm_medium=left-menu&utm_campaign=what-is-jenkins.

Selenium Documentation, Introduction. N.d. Dokumentaatio Seleniumin www-sivuilta. Viitattu 12.7.2017. http://www.seleniumhq.org/docs/01_introducing_selenium.jsp.

Smart, J. 2011. Jenkins: The Definitive Guide. N.d. O'Reilly Media. Opas Bogotobogo:n www-sivulta. Viitattu 20.8.2017. http://www.bogotobogo.com/DevOps/Jenkins/images/Intro_install/jenkins-the-definitive-guide.pdf.

Using the Page Object Pattern with Robot Framework. N.d. Artikkeli beer30.org www-sivulla. Viitattu 12.9.2017. http://www.beer30.org/using-the-page-object-pattern-with-robot-framework/.

Vogel, L. 2017. Continuous integration with Jenkins – Tutorial. Opas Vogelin www-sivuilla. Viitattu 18.7.2017. http://www.vogella.com/tutorials/Jenkins/article.html.

What is Maven?. 2017. Artikkeli Mavenin www-sivuilla. Viitattu 20.7.2017. https://maven.apache.org/what-is-maven.html.

White Box Testing. N.d. Dokumentti Software Testing Fundamentals www-sivulla. Viitattu 12.7.2017. http://softwaretestingfundamentals.com/white-box-testing/.

Liitteet

Liite 1. Testitapausten kuvaukset

Project Name:	Verkkokauppa				
Test suite name:	Verkkokauppa Maintenance & Smoke Tests				
Created by:	Santeri Moilanen & Veetu Salminen				
Date of creation	1.3.2018				
Date	Comment				
2.4.2018	First version of this document				
Notes					
Test steps are descriped at high level!					
Test Case ID/Name	Test Scenario	Pre-Condition	TEST STEPS	EXCEPTED RESULT	POST CONDITION
1.Should be able to login	Verify login to the page	1. Need a valid account to do the login	1. Open "Login" windows, 2. Enter valid "Username", 3. Enter valid "Password", 4. Click "Kirjaudu" Button	Successfull login	"Oma tili" - buttton is shown in TopNav
2.Should be able to buy a regular item	Verify that user is able to buy regular item	1. Need a valid account to do the login 2. There is items available in "hevonen" category	1. Same as Case ID:1, 2. Open Shopping Cart, 3. Clear Shopping Cart, 4. Search for Single Product "hevonen", 5. Add Hevonen to cart, 6. Proceed to Shopping Cart, 7. Proceed to Checkout	Successfull Proceeding to Checkout	Added item is at shopping cart. There is zippingmethod selected and user is proceed to checkout page
3. Should be able to buy and verify metric item	Verify that user is able to buy metric item	1. Need a valid account to do the login 2. There is item "Vallila Poppamies 150 cm verhokangas" available	1. Same as Case ID:1, 2. Open Shopping Cart, 3. Clear Shopping Cart, 4. Search for Single Product "Vallila Poppamies 150 cm verhokangas", 5. Add item to cart, 6. Proceed to Shopping Cart, 7. Proceed to Checkout	Successfull Proceeding to Checkout	Added item is at shopping cart. There is zippingmethod selected and user is proceed to checkout page
4. Should be able to purchase with bulk discount	Verify that user is able to buy with bulk discount	1. Need a valid account to do the login 2. There is item "kampamies-louna-taskukampa" available	1. Same as Case ID:1, 2. Open Shopping Cart, 3. Clear Shopping Cart, 4. Search for Single Product "kampamies-louna-taskukampa", 5. Add item to cart, 6. Proceed to Shopping Cart, 7. Proceed to Checkout	Successfull Proceeding to Checkout	Added item is at shopping cart. There is zippingmethod selected and user is proceed to checkout page
5. Should be able to verify categories	Verify that normal, brand shopit and campaign categories are	1. Categories are set from store management tools	1.Open CategoryPage "normal_cat" 2.Verify category, "normal_cat" 3.Open CategoryPage "Brand_cat" 4.Verify category, "Brand cat" 5.Open CategoryPage "Shopit_cat" 6.Verify category, "Shopit"	All categories are showing properly	-

6. Should be able to search spareparts by car model	Verify that user is able to search sparepart s to random car	-	1. Go to spareparts page, 2. Set Random car manufacturer, model and type, 3. Make search, 4. Search Should give some results	Successfully Proceeding to spareparts search results page	There are spareparts in search results page
7. Should be able to buy toyota sparepart	Verify that user is able to buy spesific sparepart s for	1. Spareparts search returns spareparts for spesific car model	1. Go to spareparts page 2. Set car manufacturer: "Toyota", model: "COROLLA Sedan E12" and type: "1.4 D-4D", 3. Make search, 4.Search for "Suodatin" 5. Select	Successfull Proceeding to Shoppingcart	There is one item in shoppingcart
8. Should be able to shop with Mini Shopping Cart	Verify that Mini Shopping cart is working properly	1. There is items "kron-basic-tiskiharja" and "airam-led-decor-5,5w-e14-kynttilä-lamppu" available	1. Go to "kron-basic-tiskiharja"-page 2. Add item to cart 3. Go to "airam-led-decor-5,5w-e14-kynttilä-lamppu"-page 4. Add item to cart 5. Delete all from mini shopping cart 6. Verify Mini Shopping Cart is Empty 7. Add item to cart 8. Proceed to shoppingcart page	Successfull Proceeding to Shoppingcart	There is one item in shoppingcart